# Microarchitectural Attacks: Meltdown and Spectre

**Daniel Gruss**

April 21, 2018

Graz University of Technology

# Whoami

- Daniel Gruss
- Post-Doc @ Graz University of Technology
- Twitter: `@lavados`
- Email: `daniel.gruss@iaik.tugraz.at`

# The Fallout

You realize it is something big when...

# The Fallout

You realize it is something big when...

- it is in the news, all over the world


FOX
BUSINESS
WASHINGTON, D.C.
WASHINGTON, D.C.
WINTER STORM
NEWS
› ALERT
INTEL REVEALS DESIGN FLAW THAT
COULD ALLOW HACKERS TO ACCESS DATA
FOX
BUSINESS
NETWORK
@FOXBUSINESS


DEVELOPING STORY
COMPUTER CHIP FLAWS IMPACT BILLIONS OF DEVICES
LIVE
CNN
NEWS STREAM


GLOBAL
COMPUTER CHIP SCARE
The bugs are known as 'Spectre' and 'Meltdown'
BBC WORLD NEWS
£:HK$ 10.58
EURO:£ 0.891


SECURITY FLAW REVEALED
Intel (Prev)
45.26
-1.59
[-3.39%]
Intel (After Hours)
44.85
-0.41
[-0.91%]
CAPITAL CONNECTION
SHROUT: ISSUE NOT UNIQUE TO INTEL, BUT IT'S AFFECTED THE MOST
CNBC

# The Fallout

You realize it is something big when...

- it is in the news, all over the world
- you get a Wikipedia article in multiple languages

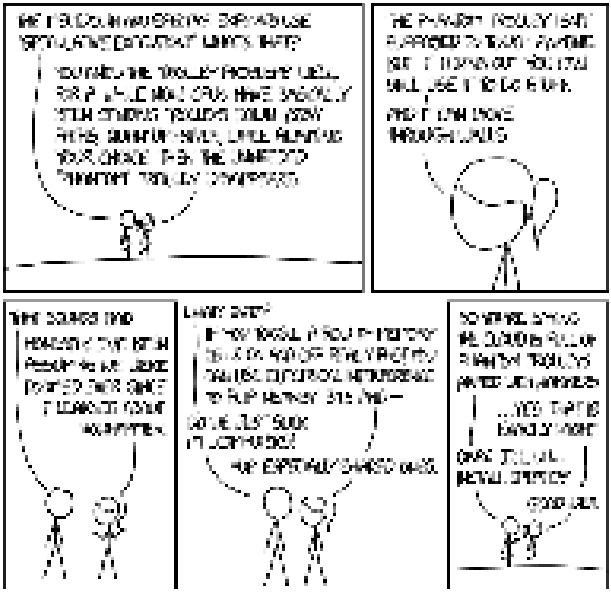

Wikipedia
The Free Encyclopedia
Article
Talk
Read
Edit
View history
Meltdown (security vulnerability)
From Wikipedia, the free encyclopedia
Meltdown is a hardware vulnerability affecting Intel x86 microprocessors and some ARM-based microprocessors.[1][2][3] It allows a rogue process to read all memory, even when it is not authorized to do so.
Meltdown affects a wide range of systems. At the time of disclosure, this included all devices running any but the most recent and patched versions of iOS,[4] Linux[5][6], macOS,[4] or Windows. Accordingly, many servers and cloud services were impacted,[7] as well as a potential majority of smart devices and embedded devices using ARM based processors (mobile devices, smart TVs and others), including a wide range of networking equipment. A purely software workaround to Meltdown has been assessed as slowing computers between 5 and 30 percent in certain specialized workloads,[8] although companies responsible for software correction of the exploit are reporting minimal impact from general benchmark testing.[9]
Meltdown was issued a Common Vulnerabilities and Exposures ID of CVE-2017-5754 et., also known as Rogue Data Cache Load,[7] in January 2018. It was disclosed in conjunction with another exploit, Spectre, with which it shares some, but not all characteristics. The Meltdown and Spectre vulnerabilities are considered "catastrophic"
MELTDOWN
The logo used by the team that discovered the vulnerability

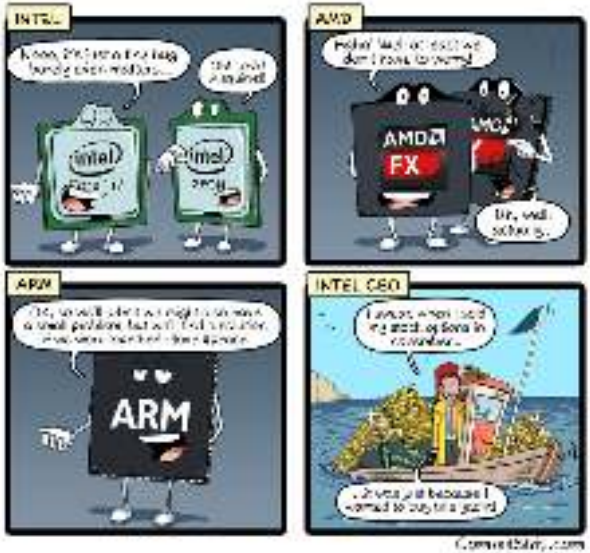

Not logged in Talk Contributions Create account Log in
Article Talk
Read Edit View history
Search Wikipedia
WIKIPEDIA
The Free Encyclopedia
Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store
Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page
Tools
What links here
Related changes
Upload file
Spectre (security vulnerability)
From Wikipedia, the free encyclopedia
Spectre is a vulnerability that affects modern microprocessors that perform branch prediction.[1][2][3] On most processors, the speculative execution resulting from a branch misprediction may leave observable side effects that may reveal private data to attackers. For example, if the pattern of memory accesses performed by such speculative execution depends on private data, the resulting state of the data cache constitutes a side channel through which an attacker may be able to extract information about the private data using a timing attack.[4][5][6]
Two Common Vulnerabilities and Exposures IDs related to Spectre, CVE-2017-5753 (bounds check bypass) and CVE-2017-5715 (branch target injection), have been issued.[7] JIT engines used for JavaScript were found vulnerable. A website can read data stored in the browser for another website, or the browser's memory itself.[8]
SPECTRE
A logo created for the vulnerability, featuring a ghost with a branch
Several procedures to help protect home computers and related devices from the Spectre (and Meltdown) security vulnerabilities have been published.[9][10][11][12] Spectre patches have been reported to significantly slow down performance, especially on older computers; on the newer 8th generation Core platforms, benchmark performance drops of 2–14 percent have been measured.[13] Meltdown patches may also produce performance loss.[10][14][15] On January 18, 2018, unwanted reboots, even for newer Intel chips, due to

# The Fallout

You realize it is something big when...

- it is in the news, all over the world
- you get a Wikipedia article in multiple languages
- there are comics, including xkcd




INTEL
intel
intel
AMD
AMD
FX
ARM
ARM
INTEL CEO

# The Fallout

You realize it is something big when...

- it is in the news, all over the world
- you get a Wikipedia article in multiple languages
- there are comics, including xkcd
- you get a lot of Twitter follower after Snowden mentioned you

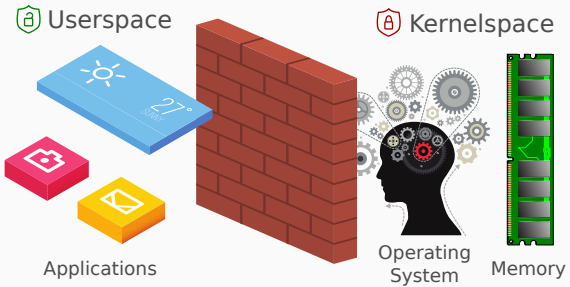
Edward Snowden

You may have heard about @intel's horrific Meltdown bug. But have you watched it in action? When your computer asks you to apply update this month, don't click "not now." [via spectreattack.com & @misc0110]

# The Wall

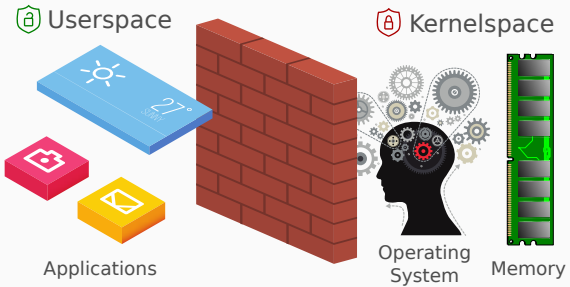
# The Core of Meltdown/Spectre

- Kernel is isolated from user space



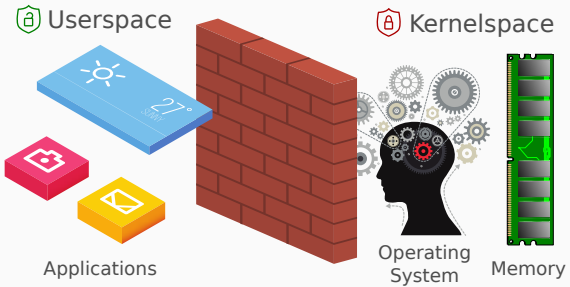
# The Core of Meltdown/Spectre

- Kernel is isolated from user space
- This isolation is a combination of hardware and software



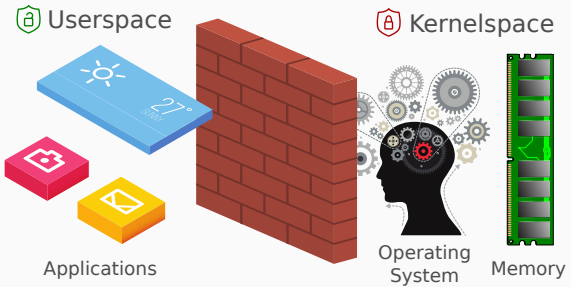
# The Core of Meltdown/Spectre

- Kernel is isolated from user space
- This isolation is a combination of hardware and software
- User applications cannot access anything from the kernel



# The Core of Meltdown/Spectre

- Kernel is isolated from user space
- This isolation is a combination of hardware and software
- User applications cannot access anything from the kernel
- There is only a well-defined interface → syscalls



# 1337 4242

## FOOD CACHE

**Revolutionary** concept!

Store your food at home, never go to the grocery store during cooking.

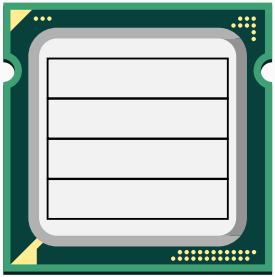
Can store **ALL** kinds of food.

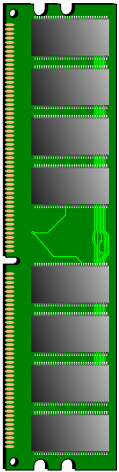
ONLY TODAY INSTEAD OF ~~$1,300~~

**$1,299**

ORDER VIA PHONE: +555 12345

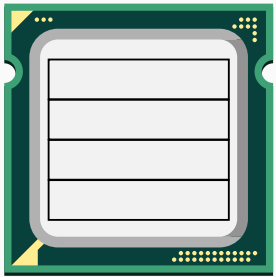
```
printf("%d", i);
printf("%d", i);
```

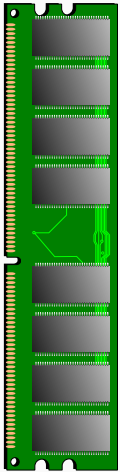




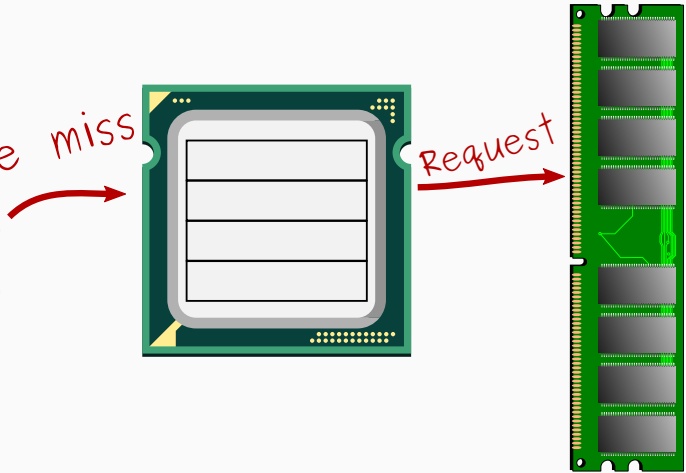
```
printf("%d", i);
printf("%d", i);
```

Cache miss

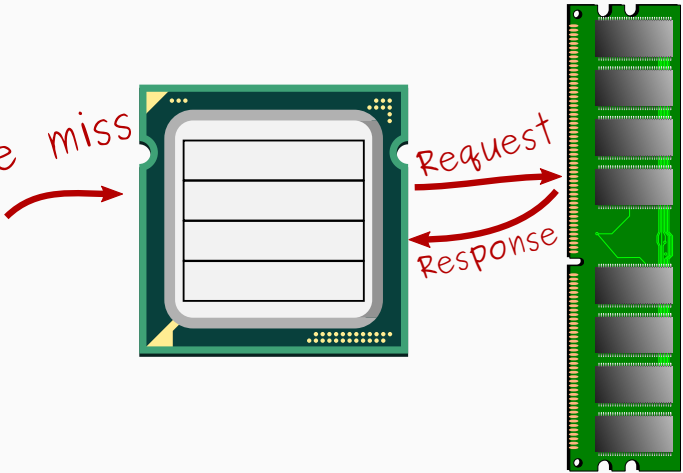




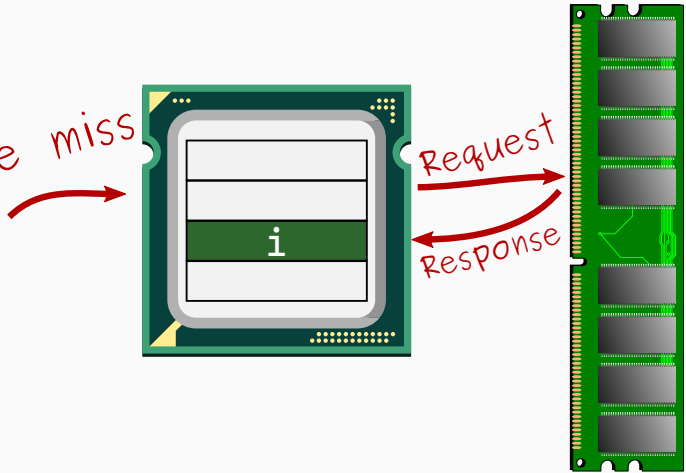
```
printf("%d", i);
printf("%d", i);
```

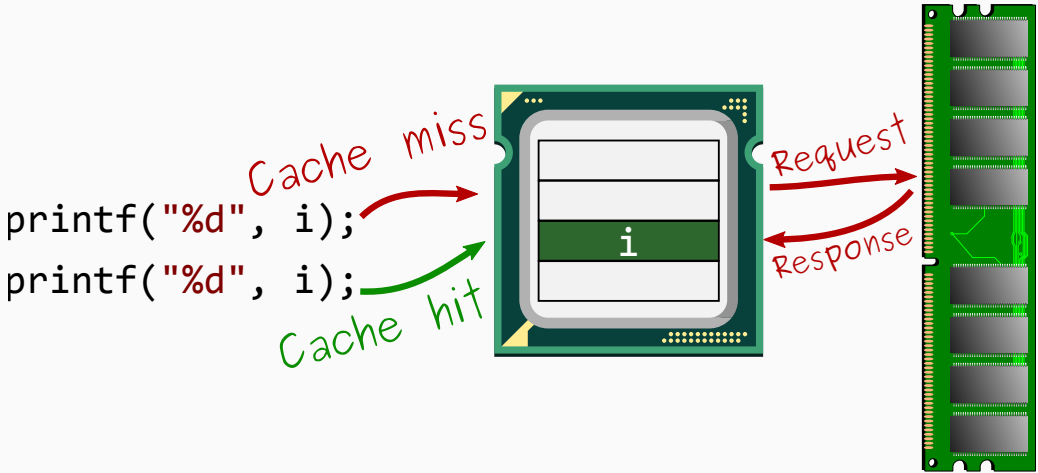


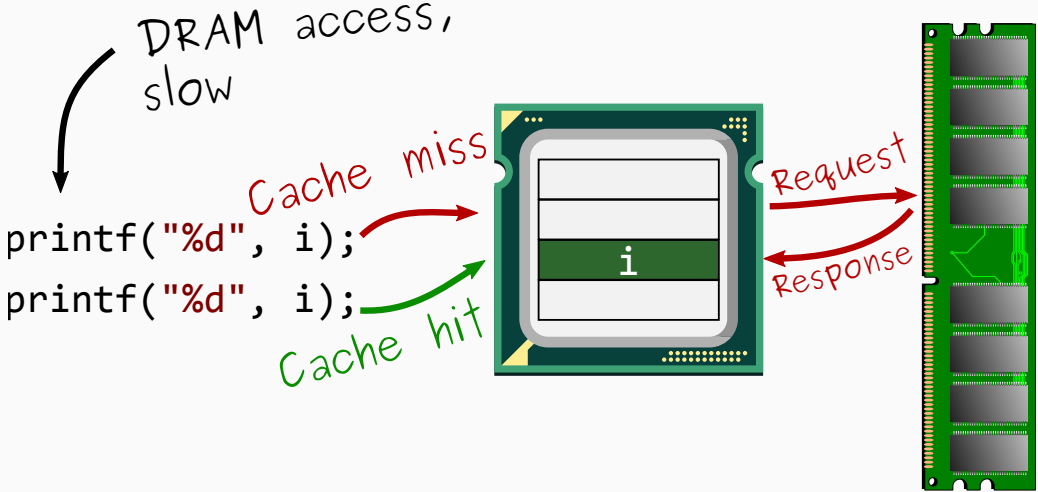
```
printf("%d", i);
printf("%d", i);
```

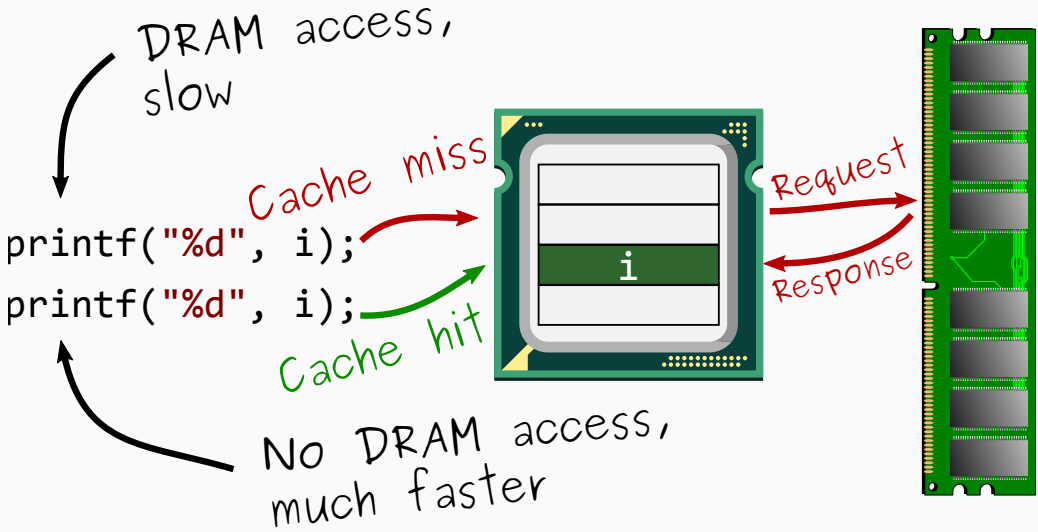


```
printf("%d", i);
printf("%d", i);
```




printf("%d", i);
printf("%d", i);
Cache miss
Cache hit
i
Request
Response


DRAM access, slow
printf("%d", i);
printf("%d", i);
Cache miss
Cache hit
i
Request
Response

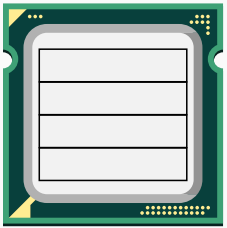

DRAM access, slow
printf("%d", i);
printf("%d", i);
Cache miss
Cache hit
i
Request
Response
No DRAM access, much faster

Shared Memory

ATTACKER

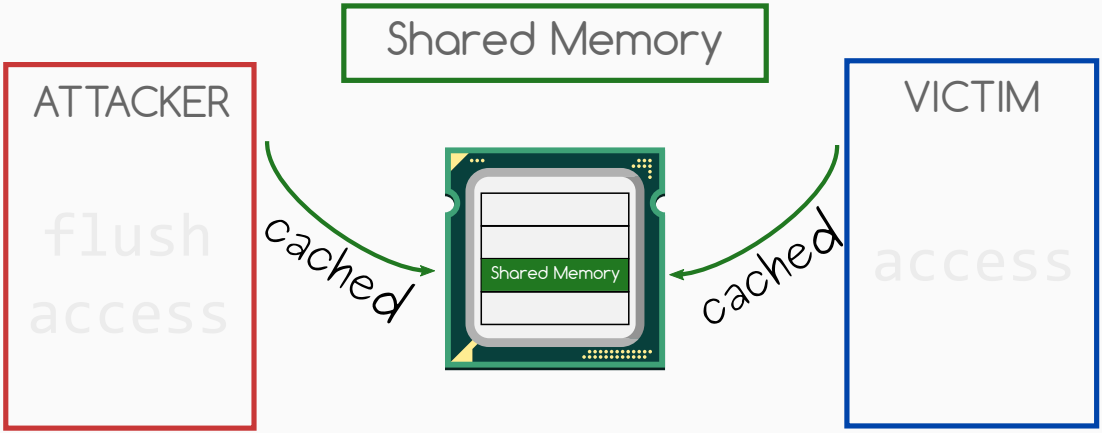
flush

access



VICTIM

access

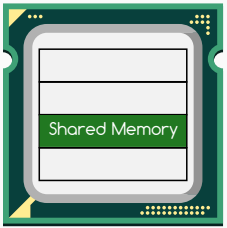

Shared Memory
ATTACKER
flush
access
cached
Shared Memory
cached
VICTIM
access

Shared Memory

ATTACKER

flush

access




VICTIM

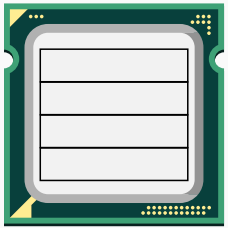
access

Shared Memory

ATTACKER

flush

access



VICTIM

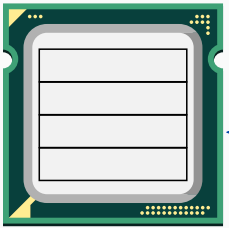
access

Shared Memory

ATTACKER

flush

access



VICTIM

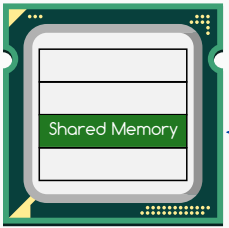
access

Shared Memory

ATTACKER

flush

access




VICTIM

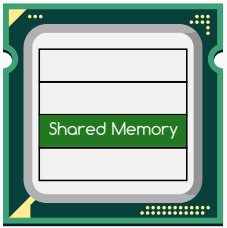
access

Shared Memory

ATTACKER

flush

access




VICTIM

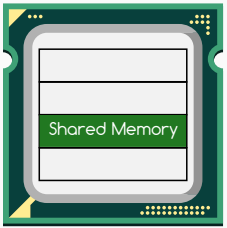
access

Shared Memory

ATTACKER

flush

access

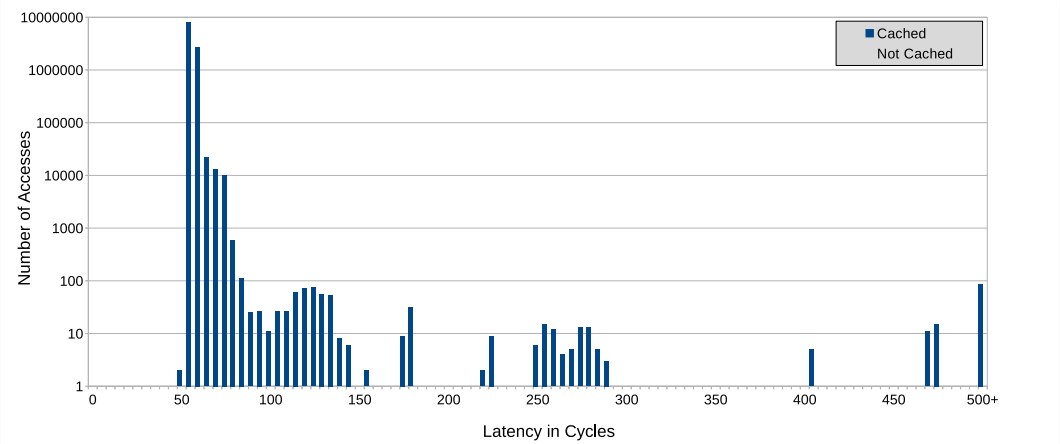



VICTIM

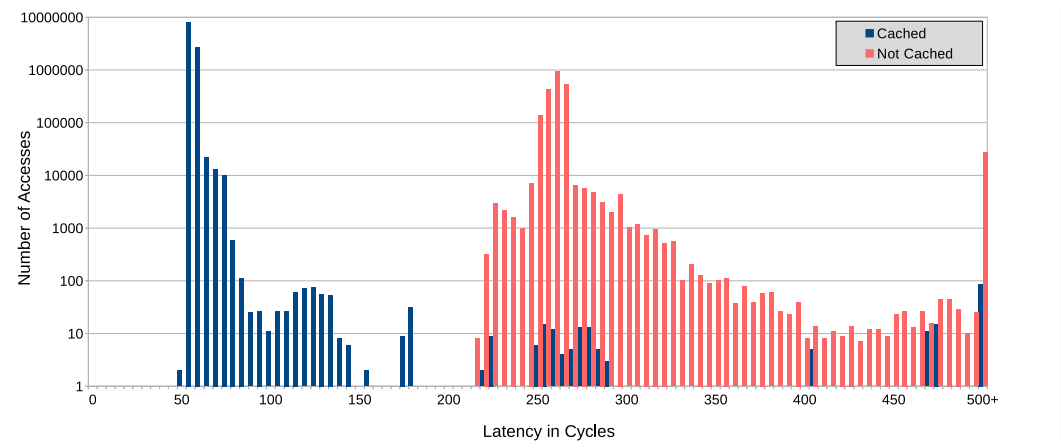
access

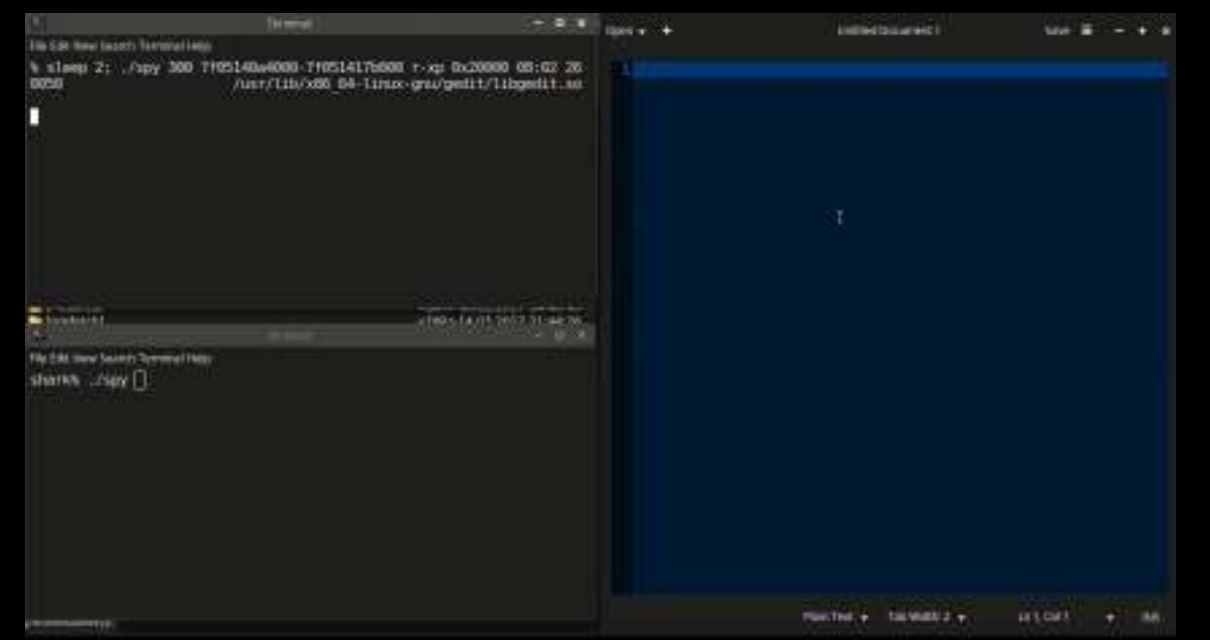
fast if victim accessed data,
slow otherwise

# Memory Access Latency

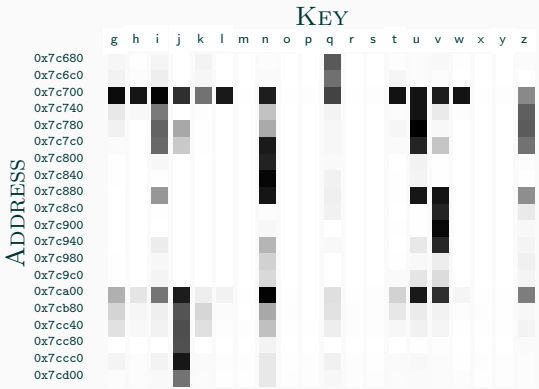


# Memory Access Latency






Cache Template Attack Demo


KEY
g h i j k l m n o p q r s t u v w x y z
ADDRESS
0x7c680
0x7c6c0
0x7c700
0x7c740
0x7c780
0x7c7c0
0x7c800
0x7c840
0x7c880
0x7c8c0
0x7c900
0x7c940
0x7c980
0x7c9c0
0x7ca00
0x7cb80
0x7cc40
0x7cc80
0x7ccc0
0x7cd00

# Out-of-order Execution

7. Serve with cooked and peeled potatoes

# Wait for an hour

# Wait for an hour

# LATENCY

1. Wash and cut vegetables

2. Pick the basil leaves and set aside

3. Heat 2 tablespoons of oil in a pan

4. Fry vegetables until golden and softened

1. Wash and cut vegetables

2. Pick the basil leaves and set aside

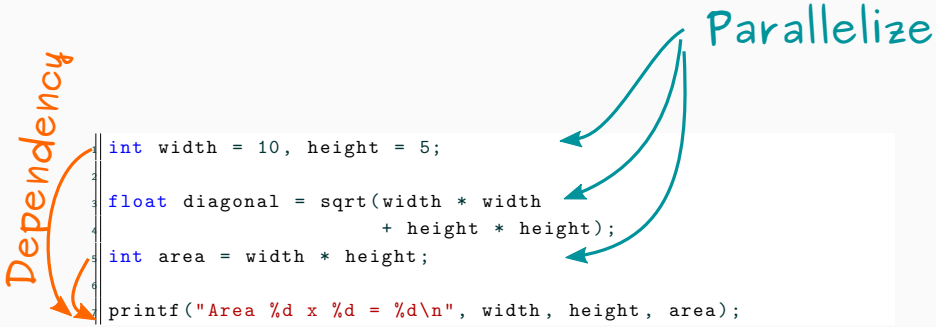
3. Heat 2 tablespoons of oil in a pan

4. Fry vegetables until golden and softened

Dependency

Parallelize

```
1 int width = 10, height = 5;
2
3 float diagonal = sqrt(width * width
4                       + height * height);
5 int area = width * height;
6
7 printf("Area %d x %d = %d\n", width, height, area);
```


Parallelize
Dependency
1 int width = 10, height = 5;
2
3 float diagonal = sqrt(width * width
4 + height * height);
5 int area = width * height;
6
7 printf("Area %d x %d = %d\n", width, height, area);

```
1 char data = *(char*)0xffffffff81a000e0;
2 printf("%c\n", data);
```

```
1 char data = *(char*)0xffffffff81a000e0;
2 printf("%c\n", data);
```

```
1 segfault at ffffffff81a000e0 ip 0000000000400535
2       sp 00007ffce4a80610 error 5 in reader
```

```
1 char data = *(char*)0xffffffff81a000e0;
2 printf("%c\n", data);
```

```
1 segfault at ffffffff81a000e0 ip 0000000000400535
2       sp 00007ffce4a80610 error 5 in reader
```

- Kernel addresses are not accessible

```
1 char data = *(char*)0xffffffff81a000e0;
2 printf("%c\n", data);
```

```
1 segfault at ffffffff81a000e0 ip 0000000000400535
2       sp 00007ffce4a80610 error 5 in reader
```

- Kernel addresses are not accessible
- Are privilege checks also done when executing instructions out of order?

- Adapted code

```
1 *(volatile char*)0;
2 array[84 * 4096] = 0; // unreachable
```

- Adapted code

```
1 *(volatile char*)0;
2 array[84 * 4096] = 0; // unreachable
```

- Static code analyzer is not happy

```
1 warning: Dereference of null pointer
2         *(volatile char*)0;
```

- Flush+Reload over all pages of the array


- "Unreachable" code line was actually executed

- Flush+Reload over all pages of the array


- "Unreachable" code line was actually executed
- Exception was only thrown afterwards

- Combine the two things

```
char data = *(char*)0xffffffff81a000e0;
array[data * 4096] = 0;
```

- Combine the two things

```
1 char data = *(char*)0xffffffff81a000e0;
2 array[data * 4096] = 0;
```

- Then check whether any part of `array` is cached

- Flush+Reload over all pages of the array


- Index of cache hit reveals data

- Flush+Reload over all pages of the array

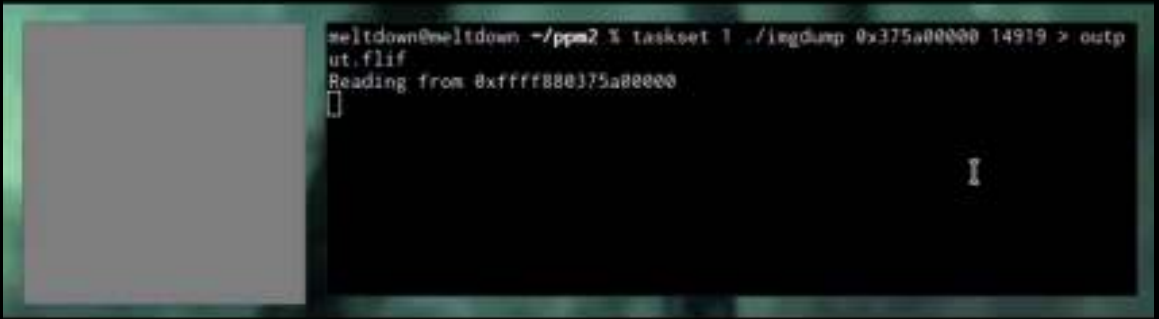

- Index of cache hit reveals data
- Permission check is in some cases not fast enough


Terminal

File Edit View Search Terminal Help

```
mschwarz@lab06:~/Documents$
```


CAN YOU
ENHANCE THAT


meltdown@meltdown ~/ppm2 % taskset 1 ./imgdump 0x375a00000 14919 > outp
ut.flif
Reading from 0xffff880375a00000

# Leaking Passwords from your Password Manager


AND IN OTHER NEWS...
MELTDOWN
WE'RE ALL DOOMED, SANDRA.
HOW ABOUT THE WEATHER?

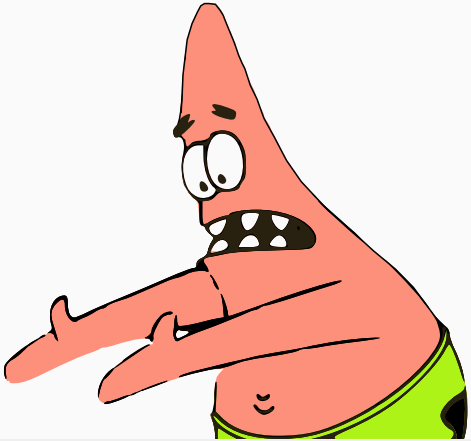
# Not so fast...

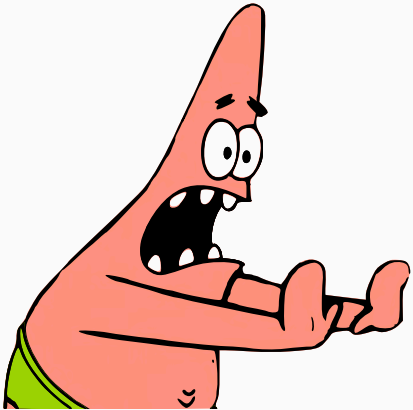
# Take the kernel addresses...

- Kernel addresses in user space are a problem

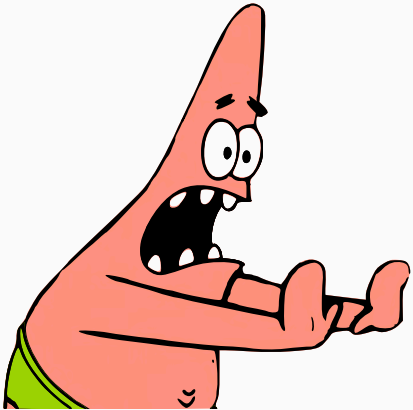
# Take the kernel addresses...

- Kernel addresses in user space are a problem
- Why don't we take the kernel addresses...





- ...and remove them if not needed?



- ...and remove them if not needed?
- User accessible check in hardware is not reliable

# Idea

- Let's just unmap the kernel in user space

# Idea

- Let's just unmap the kernel in user space
- Kernel addresses are then no longer present

# Idea

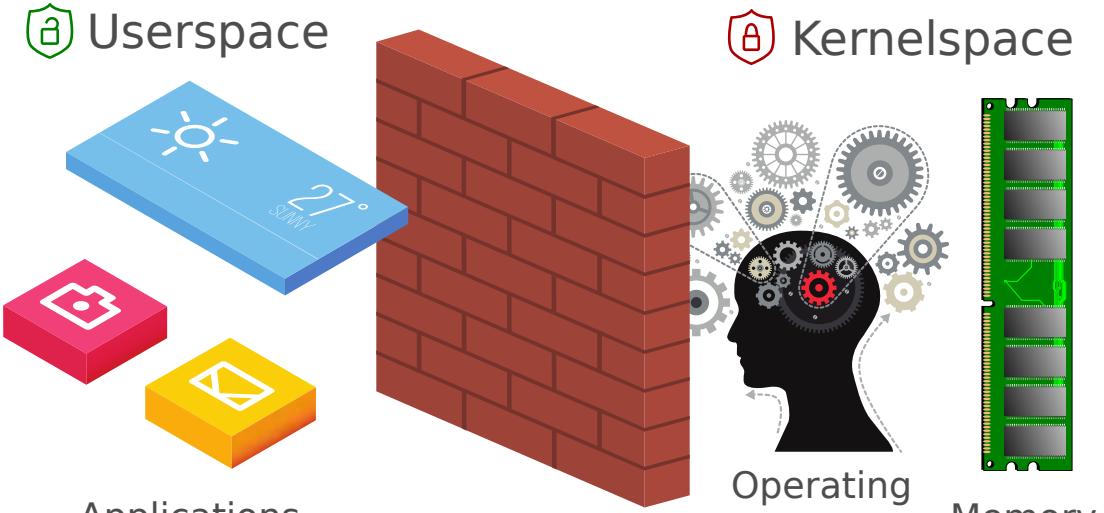
- Let's just unmap the kernel in user space
- Kernel addresses are then no longer present
- Memory which is not mapped cannot be accessed at all


Kernel Address Isolation to have Side channels Efficiently Removed

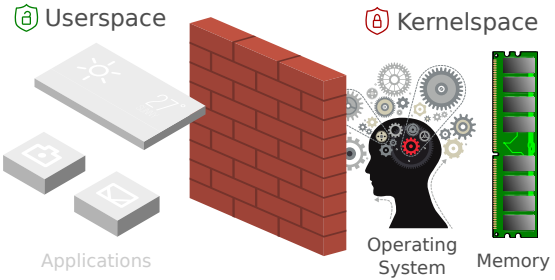
**KAISER** /ˈkʌɪzə/

1. [german] Emperor, ruler of an empire
2. largest penguin, emperor penguin

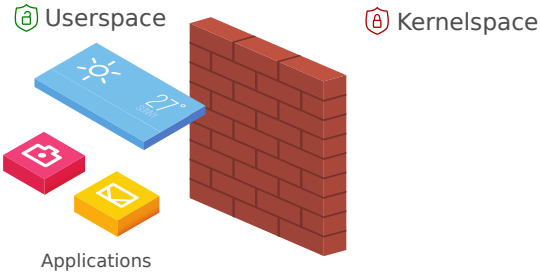
**K**ernel **A**ddress **I**solation to have **S**ide channels **E**fficiently **R**emoved

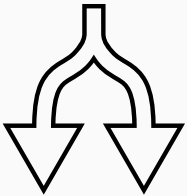

Userspace
Kernelspace
27°
Applications
Operating System
Memory

# Kernel View

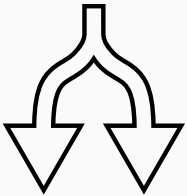



# User View

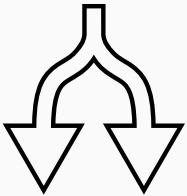


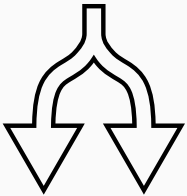
- We published KAISER in July 2017



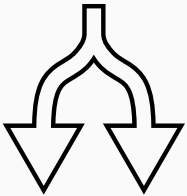
- We published KAISER in July 2017
- Intel and others improved and merged it into Linux as KPTI (Kernel Page Table Isolation)



- We published KAISER in July 2017
- Intel and others improved and merged it into Linux as KPTI (Kernel Page Table Isolation)
- Microsoft implemented similar concept in Windows 10



- We published KAISER in July 2017
- Intel and others improved and merged it into Linux as KPTI (Kernel Page Table Isolation)
- Microsoft implemented similar concept in Windows 10
- Apple implemented it in macOS 10.13.2 and called it "Double Map"

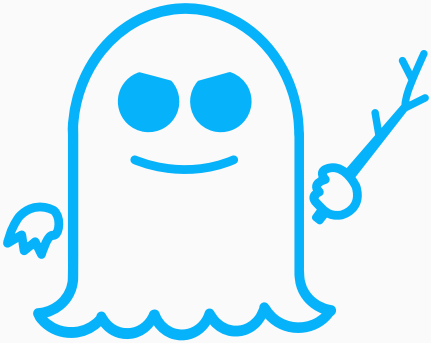


- We published KAISER in July 2017
- Intel and others improved and merged it into Linux as KPTI (Kernel Page Table Isolation)
- Microsoft implemented similar concept in Windows 10
- Apple implemented it in macOS 10.13.2 and called it "Double Map"
- All share the same idea: switching address spaces on context switch

MELTDOWN

SPECTRE

MELTDOWN

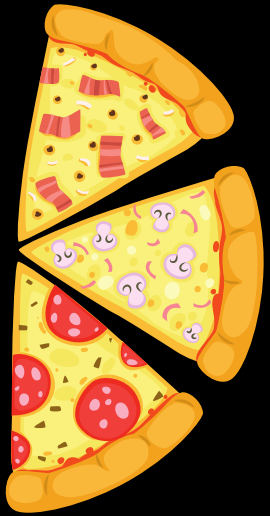


SPECTRE

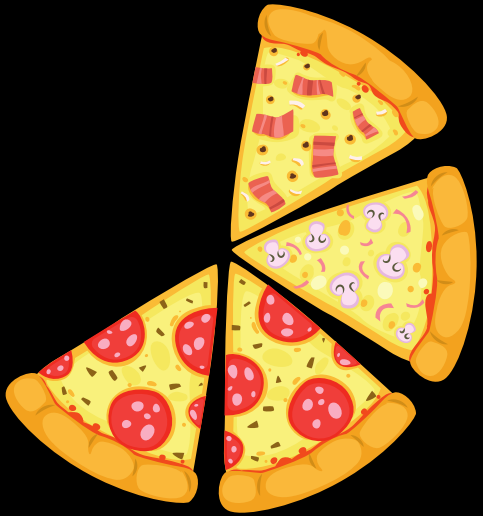

PIZZA
SPECIAL RECIPES

# Prosciutto

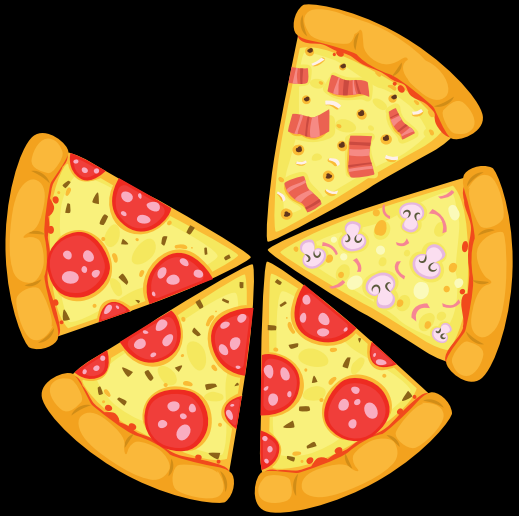
# Funghi



# Diavolo



# Diavolo



# Diavolo

# Diavolo


»A table for 6 please«

# Speculative Cooking


»A table for 6 please«


PIZZA
SPECIAL RECIPES


PIZZA
SPECIAL RECIPES


Pizza


PIZZA
SPECIAL RECIPES


Pizza


PIZZA
SPECIAL RECIPES


Pizza

# What does Spectre do?

- Mistrains branch prediction

# What does Spectre do?

- Mistrains branch prediction
- CPU speculatively executes code which should not be executed

# What does Spectre do?



- Mistrains branch prediction
- CPU speculatively executes code which should not be executed
- Can also mistrain indirect calls

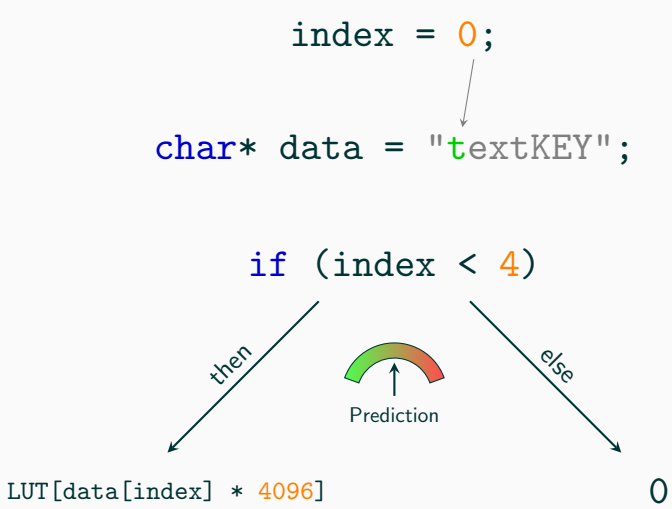
# What does Spectre do?



- Mistrains branch prediction
- CPU speculatively executes code which should not be executed
- Can also mistrain indirect calls

→ Spectre “convinces” program to execute code

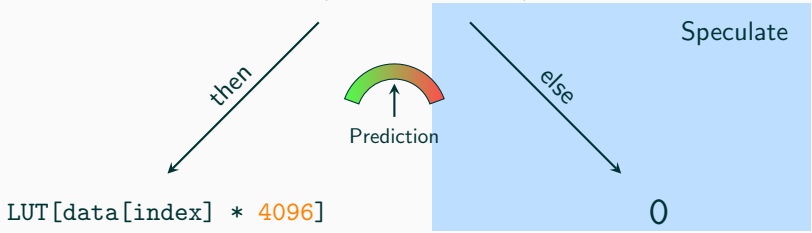
```
index = 0;

char* data = "textKEY";

if (index < 4)
```


index = 0;
char* data = "textKEY";
if (index < 4)
then
else
Prediction
LUT[data[index] * 4096]
0

```
index = 0;
```

```
char* data = "textKEY";
```

```
if (index < 4)
```

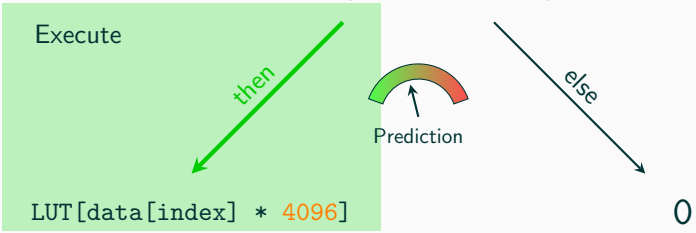


# Spectre (variant 1)

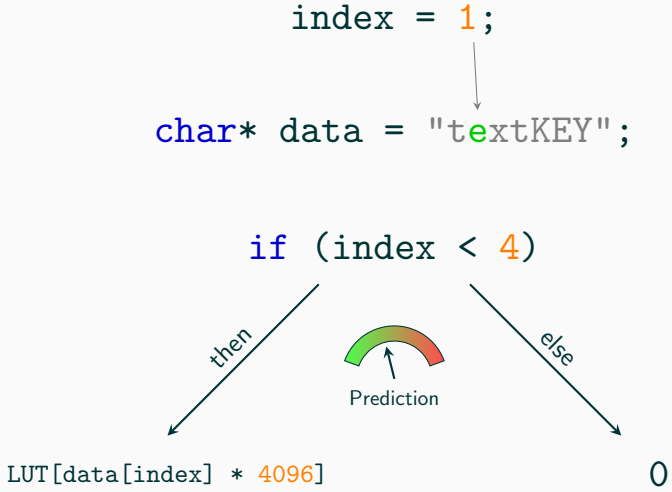
```
index = 0;
```

```
char* data = "textKEY";
```

```
if (index < 4)
```



```
index = 1;

char* data = "textKEY";

if (index < 4)
```

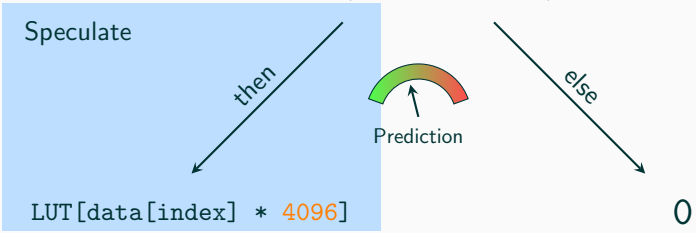

index = 1;
char* data = "textKEY";
if (index < 4)
then
else
Prediction
LUT[data[index] * 4096]
0

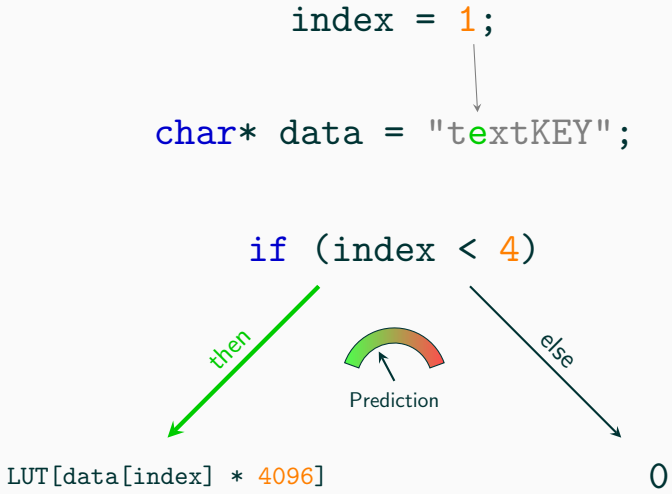
# Spectre (variant 1)

```
index = 1;
```

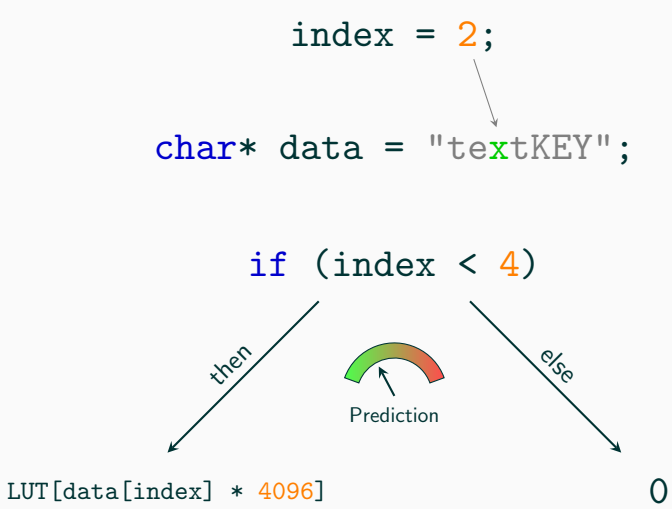
```
char* data = "textKEY";
```

```
if (index < 4)
```




index = 1;
char* data = "textKEY";
if (index < 4)
then
Prediction
else
LUT[data[index] * 4096]
0

```
index = 2;

char* data = "textKEY";

if (index < 4)
```

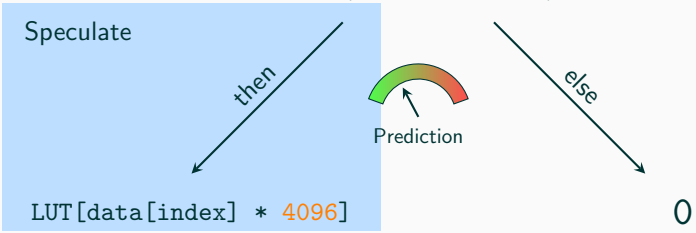

index = 2;
char* data = "textKEY";
if (index < 4)
then
else
Prediction
LUT[data[index] * 4096]
0

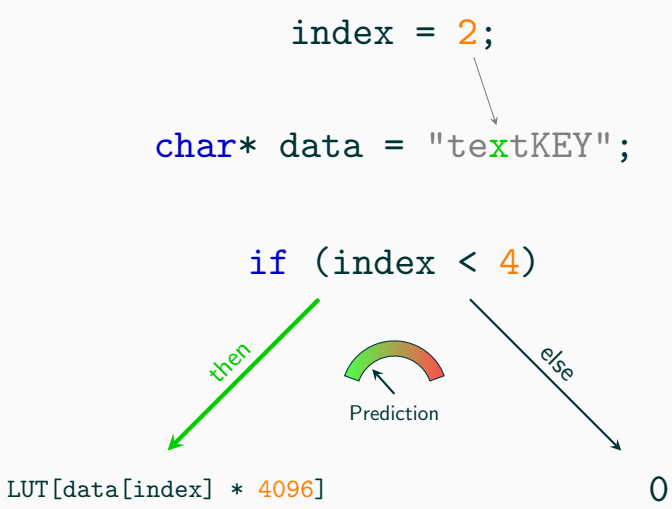
# Spectre (variant 1)

```
index = 2;
```

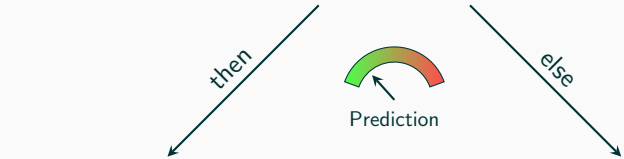
```
char* data = "textKEY";
```

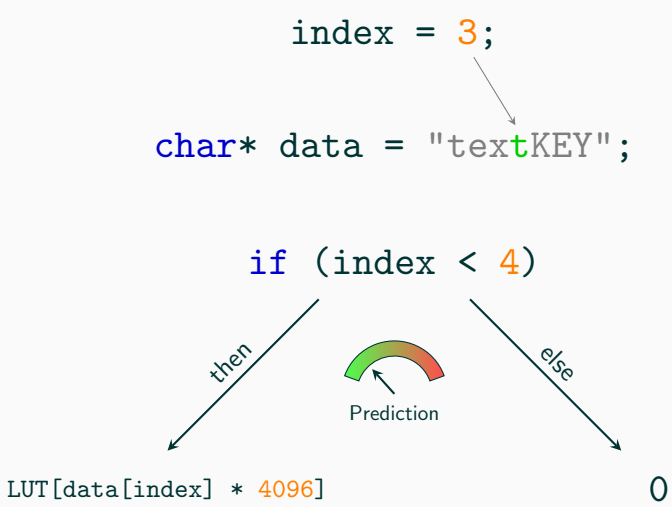
```
if (index < 4)
```




index = 2;
char* data = "textKEY";
if (index < 4)
then
Prediction
else
LUT[data[index] * 4096]
0

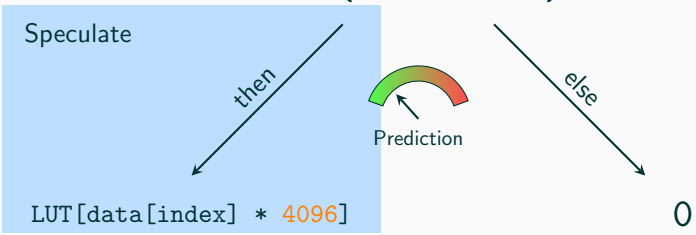
```
index = 3;

char* data = "textKEY";

if (index < 4)
```




index = 3;
char* data = "textKEY";
if (index < 4)
then
else
Prediction
LUT[data[index] * 4096]
0

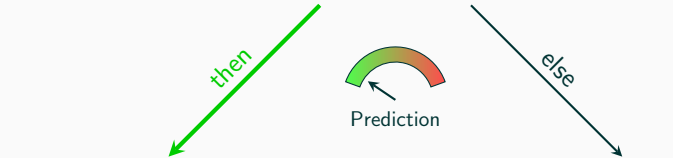
# Spectre (variant 1)

```
index = 3;
char* data = "textKEY";
if (index < 4)
```

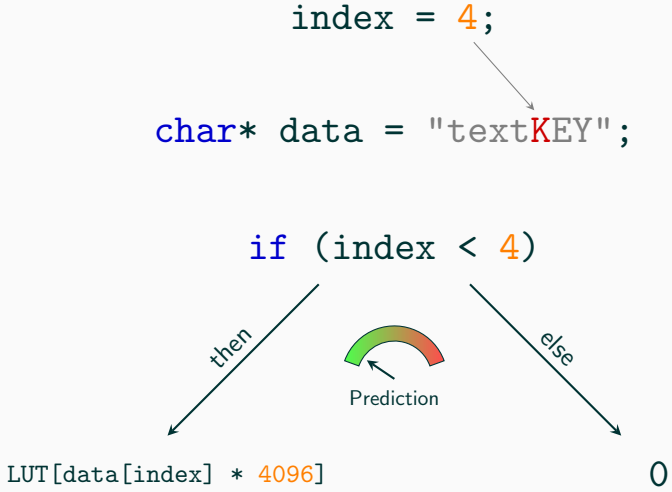


# Spectre (variant 1)

```
index = 3;
char* data = "textKEY";
if (index < 4)
```

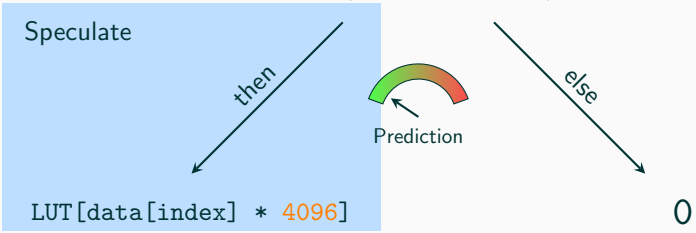


```
index = 4;

char* data = "textKEY";

if (index < 4)
```


index = 4;
char* data = "textKEY";
if (index < 4)
then
Prediction
else
LUT[data[index] * 4096]
0

# Spectre (variant 1)

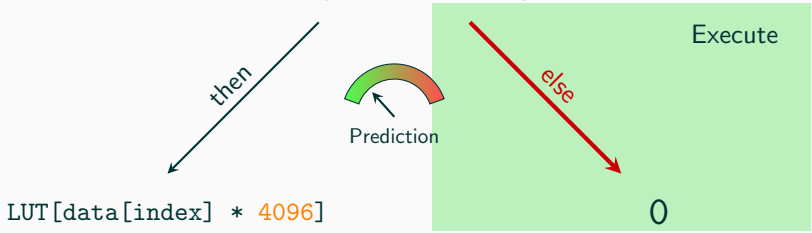
```
index = 4;

char* data = "textKEY";

if (index < 4)
```



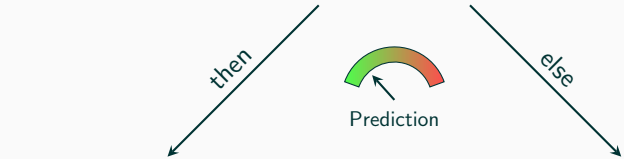
```
index = 4;
```

```
char* data = "textKEY";
```

```
if (index < 4)
```



```
index = 5;

char* data = "textKEY";

if (index < 4)
```



# Spectre (variant 1)

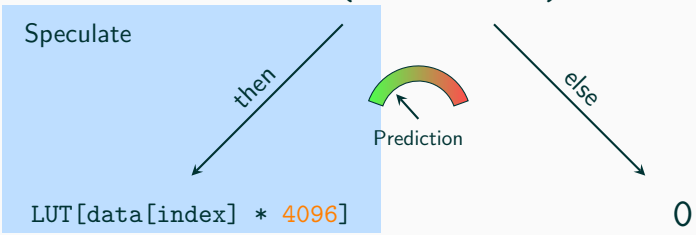
```
index = 5;

char* data = "textKEY";

if (index < 4)
```


`LUT[data[index] * 4096]`

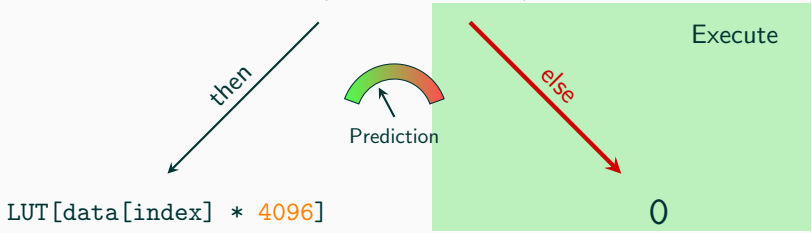
0

# Spectre (variant 1)

```
index = 5;

char* data = "textKEY";

if (index < 4)
```



```
index = 5;

char* data = "textKEY";

if (index < 4)
```

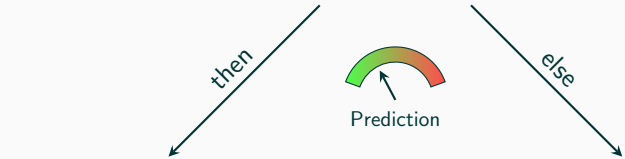


```
index = 6;

char* data = "textKEY";

if (index < 4)
```

```
index = 6;
```

```
char* data = "textKEY";
```

```
if (index < 4)
```

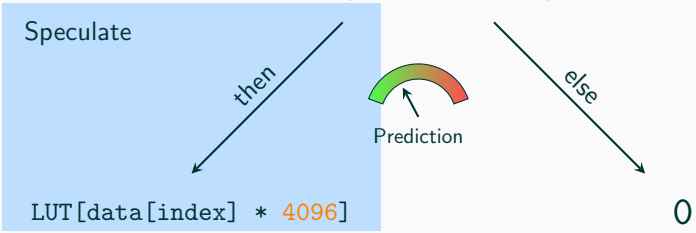



```
LUT[data[index] * 4096]
```

0

# Spectre (variant 1)

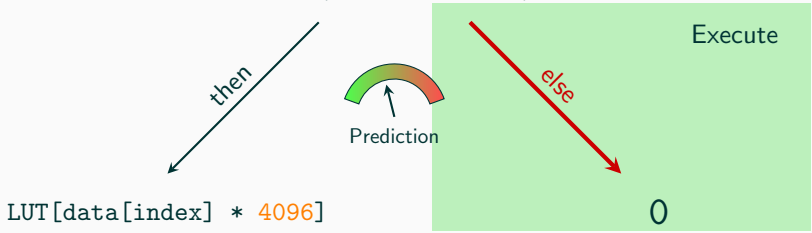
```
index = 6;

char* data = "textKEY";

if (index < 4)
```



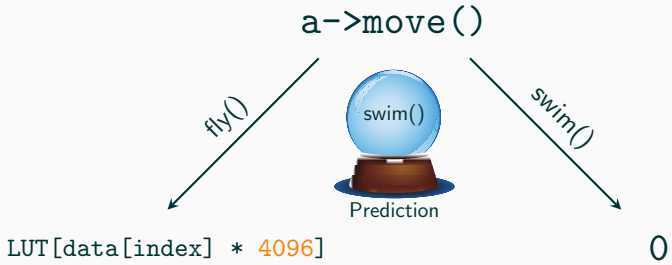
```
index = 6;
```

```
char* data = "textKEY";
```

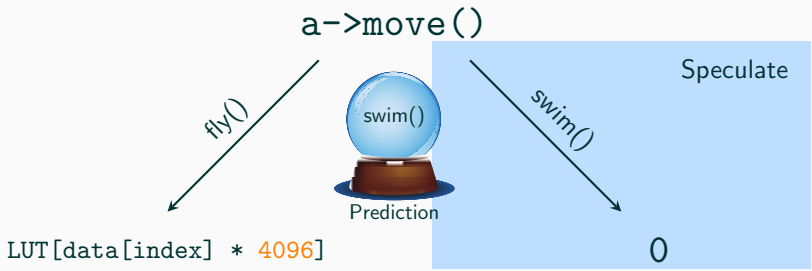
```
if (index < 4)
```



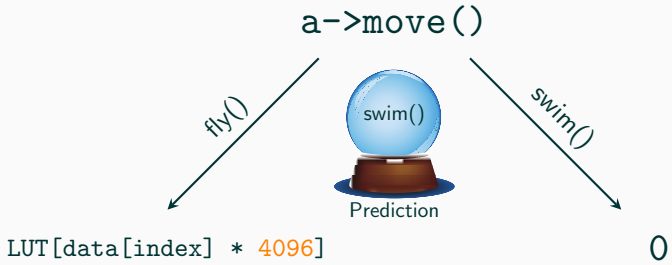
```
Animal* a = bird;
```



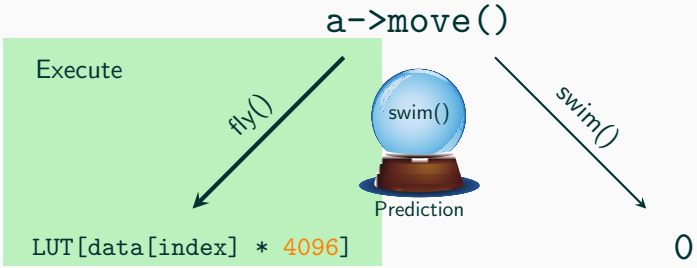
```
Animal* a = bird;
```



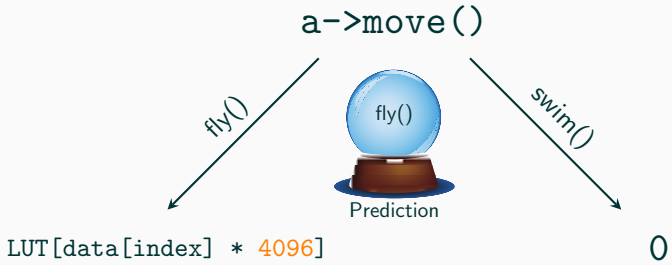
```
Animal* a = bird;
```



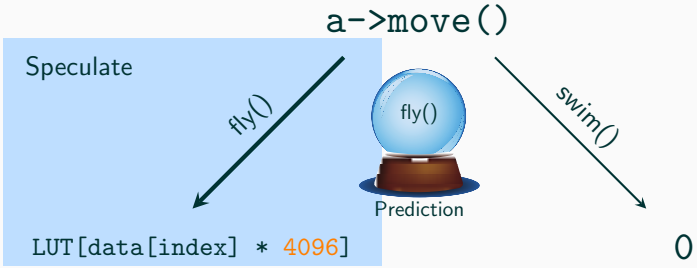
```
Animal* a = bird;
```



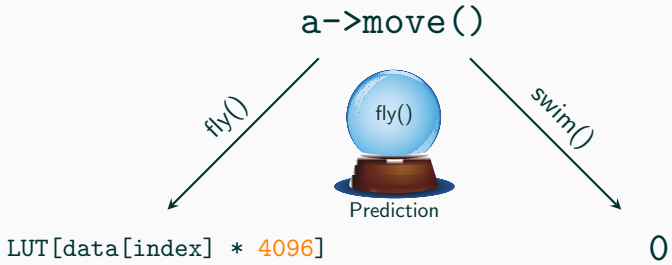
```
Animal* a = bird;
```



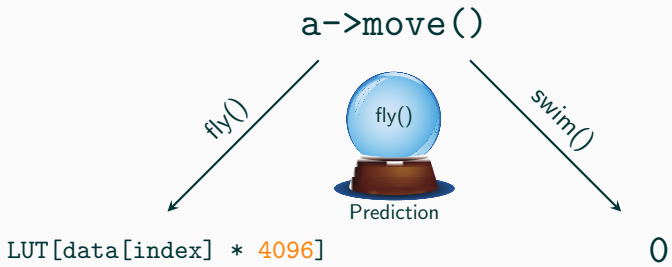
```
Animal* a = bird;
```



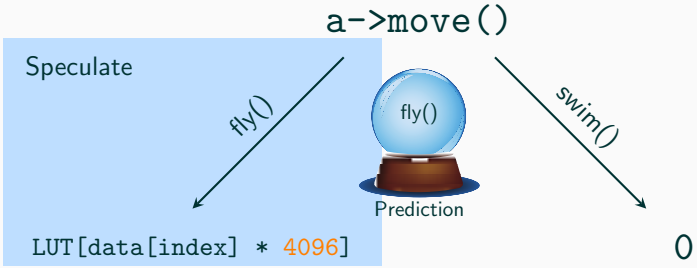
```
Animal* a = bird;
```



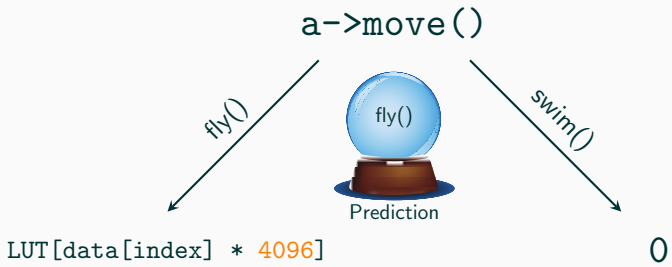
```
Animal* a = fish;
```



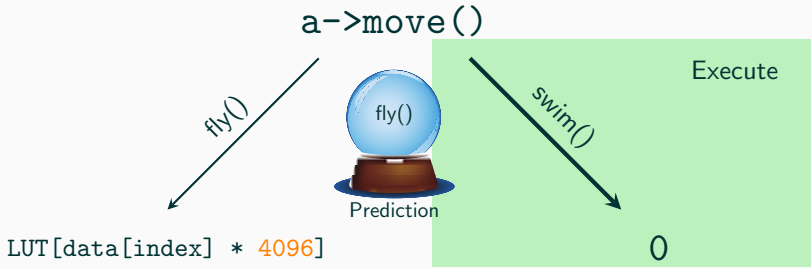
```
Animal* a = fish;
```



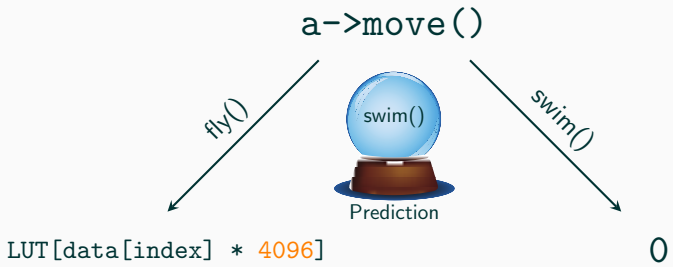
```
Animal* a = fish;
```



```
Animal* a = fish;
```



```
Animal* a = fish;
```



- Trivial approach: disable speculative execution

- Trivial approach: disable speculative execution
- No wrong speculation if there is no speculation

- Trivial approach: disable speculative execution
- No wrong speculation if there is no speculation
- Problem: massive performance hit!

- Trivial approach: disable speculative execution
- No wrong speculation if there is no speculation
- Problem: massive performance hit!
- Also: How to disable it?

- Trivial approach: disable speculative execution
- No wrong speculation if there is no speculation
- Problem: massive performance hit!
- Also: How to disable it?
- Speculative execution is deeply integrated into CPU


GHOSTBUSTERS


- Workaround: insert instructions stopping speculation


- Workaround: insert instructions stopping speculation

$\rightarrow$ insert after every bounds check


- Workaround: insert instructions stopping speculation

$\rightarrow$ insert after every bounds check

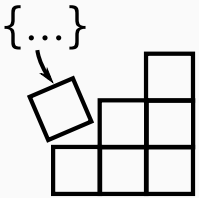
- x86: `LFENCE`, ARM: `CSDB`

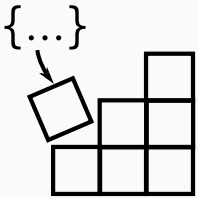
# Spectre Variant 1 Mitigations


- Workaround: insert instructions stopping speculation

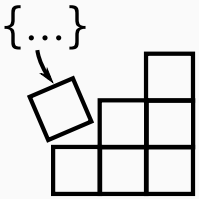
→ insert after every bounds check

- x86: `LFENCE`, ARM: `CSDB`
- Available on all Intel CPUs, retrofitted to existing ARMv7 and ARMv8

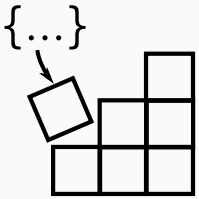




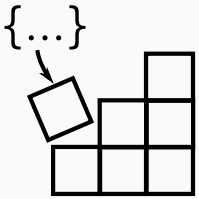

- Speculation barrier requires compiler supported




- Speculation barrier requires compiler supported
- Already implemented in GCC, LLVM, and MSVC




- Speculation barrier requires compiler supported
- Already implemented in GCC, LLVM, and MSVC
- Can be automated (MSVC) $\rightarrow$ not really reliable




- Speculation barrier requires compiler supported
- Already implemented in GCC, LLVM, and MSVC
- Can be automated (MSVC) → not really reliable
- Explicit use by programmer: `__builtin_load_no_speculate`

# Spectre Variant 1 Mitigations

```
// Unprotected

...r array[9];

int get_value(unsigned int a) {
  int tmp;

  if (a < 9) {
    tmp = array[a];
  } else {
    tmp = FAIL;
  }

  return tmp;
}
```

# Spectre Variant 1 Mitigations

```
// Unprotected

int array[N];

int get_value(unsigned int n) {
  int tmp;

  if (n < N) {
    tmp = array[n];
  } else {
    tmp = FAIL;
  }

  return tmp;
}
```

```
// Protected

int array[N];

int get_value(unsigned int n) {

  int *lower = array;
  int *ptr = array + n;
  int *upper = array + N;

  return
    __builtin_load_no_speculate
    (ptr, lower, upper, FAIL);

}
```

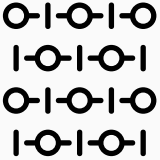

GHOSTBUSTERS


- Speculation barrier works if affected code constructs are known


- Speculation barrier works if affected code constructs are known
- Programmer has to fully understand vulnerability

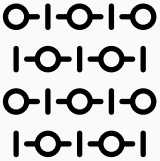

- Speculation barrier works if affected code constructs are known
- Programmer has to fully understand vulnerability
- Automatic detection is not reliable


- Speculation barrier works if affected code constructs are known
- Programmer has to fully understand vulnerability
- Automatic detection is not reliable
- Non-negligible performance overhead of barriers



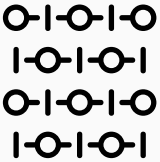
Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):



Intel released microcode updates

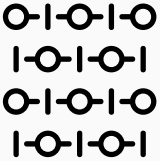
- Indirect Branch Restricted Speculation (IBRS):
  - Do not speculate based on anything before entering IBRS mode



Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):
  - Do not speculate based on anything before entering IBRS mode

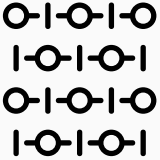
  $\rightarrow$ lesser privileged code cannot influence predictions



Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):
  - Do not speculate based on anything before entering IBRS mode

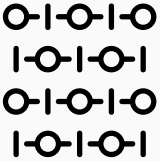
  $\rightarrow$ lesser privileged code cannot influence predictions
- Indirect Branch Predictor Barrier (IBPB):



Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):
  - Do not speculate based on anything before entering IBRS mode

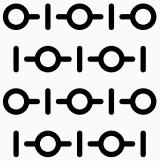
  $\rightarrow$ lesser privileged code cannot influence predictions
- Indirect Branch Predictor Barrier (IBPB):
  - Flush branch-target buffer



Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):
  - Do not speculate based on anything before entering IBRS mode

  $\rightarrow$ lesser privileged code cannot influence predictions
- Indirect Branch Predictor Barrier (IBPB):
  - Flush branch-target buffer
- Single Thread Indirect Branch Predictors (STIBP):

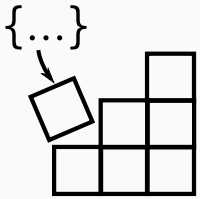


Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):
  - Do not speculate based on anything before entering IBRS mode

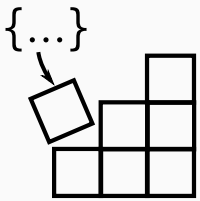
  $\rightarrow$ lesser privileged code cannot influence predictions
- Indirect Branch Predictor Barrier (IBPB):
  - Flush branch-target buffer
- Single Thread Indirect Branch Predictors (STIBP):
  - Isolates branch prediction state between two hyperthreads

Retpoline (compiler extension)



Retpoline (compiler extension)

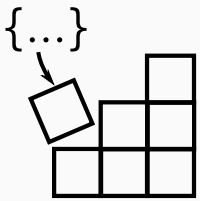
```
1       push <call_target>
2       call 1f
3     2:                    ; speculation will continue here
4       lfence              ; speculation barrier
5       jmp 2b              ; endless loop
6     1:
7       lea 8(%rsp), %rsp   ; restore stack pointer
8       ret                 ; the actual call to <call_target>
```




→ always predict to enter an endless loop

Retpoline (compiler extension)

```
1       push <call_target>
2       call 1f
3     2:                    ; speculation will continue here
4       lfence              ; speculation barrier
5       jmp 2b              ; endless loop
6     1:
7       lea 8(%rsp), %rsp   ; restore stack pointer
8       ret                 ; the actual call to <call_target>
```

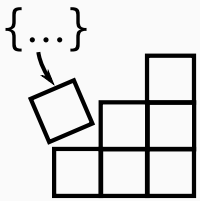



→ always predict to enter an endless loop

- instead of the correct (or wrong) target function

Retpoline (compiler extension)

```
1       push <call_target>
2       call 1f
3     2:                    ; speculation will continue here
4       lfence              ; speculation barrier
5       jmp 2b              ; endless loop
6     1:
7       lea 8(%rsp), %rsp   ; restore stack pointer
8       ret                 ; the actual call to <call_target>
```

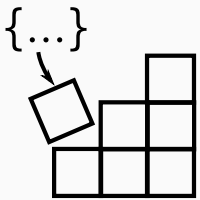



→ always predict to enter an endless loop

- instead of the correct (or wrong) target function → performance?

Retpoline (compiler extension)

```
1       push <call_target>
2       call 1f
3     2:                    ; speculation will continue here
4       lfence              ; speculation barrier
5       jmp 2b              ; endless loop
6     1:
7       lea 8(%rsp), %rsp   ; restore stack pointer
8       ret                 ; the actual call to <call_target>
```

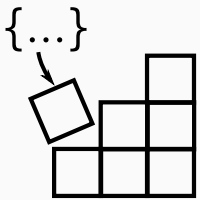



→ always predict to enter an endless loop

- instead of the correct (or wrong) target function → performance?
- On Broadwell or newer:

Retpoline (compiler extension)

```
1       push <call_target>
2       call 1f
3     2:                    ; speculation will continue here
4       lfence              ; speculation barrier
5       jmp 2b              ; endless loop
6     1:
7       lea 8(%rsp), %rsp   ; restore stack pointer
8       ret                 ; the actual call to <call_target>
```

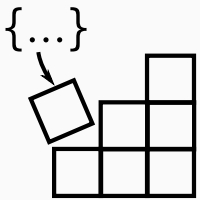



→ always predict to enter an endless loop

- instead of the correct (or wrong) target function → performance?
- On Broadwell or newer:
  - **ret** may fall-back to the BTB for prediction

Retpoline (compiler extension)

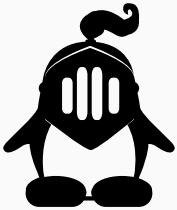
```
1       push <call_target>
2       call 1f
3     2:                    ; speculation will continue here
4       lfence              ; speculation barrier
5       jmp 2b              ; endless loop
6     1:
7       lea 8(%rsp), %rsp   ; restore stack pointer
8       ret                 ; the actual call to <call_target>
```



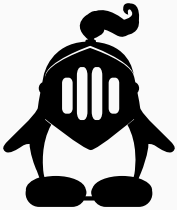

→ always predict to enter an endless loop

- instead of the correct (or wrong) target function → performance?
- On Broadwell or newer:
  - **ret** may fall-back to the BTB for prediction

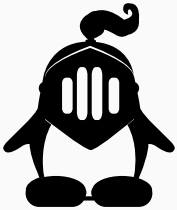
  → microcode patches to prevent that



- ARM provides hardened Linux kernel

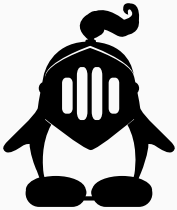


- ARM provides hardened Linux kernel
- Clears branch-predictor state on context switch



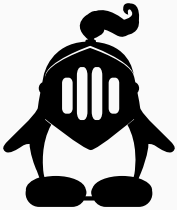
- ARM provides hardened Linux kernel
- Clears branch-predictor state on context switch
- Either via instruction (`BPIALL`)...

# Spectre Variant 2 Mitigations (Software)



- ARM provides hardened Linux kernel
- Clears branch-predictor state on context switch
- Either via instruction (`BPIALL`)...
- ...or workaround (disable/enable MMU)

# Spectre Variant 2 Mitigations (Software)



- ARM provides hardened Linux kernel
- Clears branch-predictor state on context switch
- Either via instruction (`BPIALL`)...
- ...or workaround (disable/enable MMU)
- Non-negligible performance overhead ($\approx$ 200-300 ns)

# What does not work

- Prevent access to high-resolution timer

# What does not work

- Prevent access to high-resolution timer

→ Own timer using timing thread

# What does not work

- Prevent access to high-resolution timer

→ Own timer using timing thread

- Flush instruction only privileged

# What does not work

- Prevent access to high-resolution timer

→ Own timer using timing thread

- Flush instruction only privileged

→ Cache eviction through memory accesses

# What does not work

- Prevent access to high-resolution timer

→ Own timer using timing thread

- Flush instruction only privileged

→ Cache eviction through memory accesses

- Just move secrets into secure world

# What does not work

- Prevent access to high-resolution timer

→ Own timer using timing thread

- Flush instruction only privileged

→ Cache eviction through memory accesses

- Just move secrets into secure world

→ Spectre works on secure enclaves

# Meltdown vs. Spectre

Meltdown

Spectre

# Meltdown vs. Spectre

Meltdown

- Out-of-Order Execution

Spectre

- Speculative Execution (subset of Out-of-Order Execution)

# Meltdown vs. Spectre

Meltdown

- Out-of-Order Execution
- has nothing to do with branch prediction

Spectre

- Speculative Execution (subset of Out-of-Order Execution)
- fundamentally builds on branch (mis)prediction

# Meltdown vs. Spectre

Meltdown

- Out-of-Order Execution
- has nothing to do with branch prediction
- turning off speculative execution entirely has no effect on Meltdown

Spectre

- Speculative Execution (subset of Out-of-Order Execution)
- fundamentally builds on branch (mis)prediction
- turning off speculative execution entirely would work

# Meltdown vs. Spectre

Meltdown

- Out-of-Order Execution
- has nothing to do with branch prediction
- turning off speculative execution entirely has no effect on Meltdown

→ melts down the isolation provided by the `user_accessible`-bit

Spectre

- Speculative Execution (subset of Out-of-Order Execution)
- fundamentally builds on branch (mis)prediction
- turning off speculative execution entirely would work
- has nothing to do with the `user_accessible`-bit

# Meltdown vs. Spectre

Meltdown

- Out-of-Order Execution
- has nothing to do with branch prediction
- turning off speculative execution entirely has no effect on Meltdown

→ melts down the isolation provided by the `user_accessible`-bit

- in theory: OoO not required, pipelining can be sufficient

Spectre

- Speculative Execution (subset of Out-of-Order Execution)
- fundamentally builds on branch (mis)prediction
- turning off speculative execution entirely would work
- has nothing to do with the `user_accessible`-bit
- KAISER has no effect on Spectre at all

# Meltdown vs. Spectre

Meltdown

- Out-of-Order Execution
- has nothing to do with branch prediction
- turning off speculative execution entirely has no effect on Meltdown

→ melts down the isolation provided by the `user_accessible`-bit

- in theory: OoO not required, pipelining can be sufficient
- mitigated by KAISER

Spectre

- Speculative Execution (subset of Out-of-Order Execution)
- fundamentally builds on branch (mis)prediction
- turning off speculative execution entirely would work
- has nothing to do with the `user_accessible`-bit
- KAISER has no effect on Spectre at all

# Meltdown vs. Spectre

Meltdown

Spectre

# Meltdown vs. Spectre

Meltdown

- performs illegal memory accesses $\rightarrow$ we need to take care of processor exceptions

Spectre

- performs only legal memory accesses

# Meltdown vs. Spectre

Meltdown

- performs illegal memory accesses $\rightarrow$ we need to take care of processor exceptions
  - exception handling

Spectre

- performs only legal memory accesses
  - has nothing to do with exception handling

# Meltdown vs. Spectre

Meltdown

- performs illegal memory accesses $\rightarrow$ we need to take care of processor exceptions
  - exception handling
  - exception suppression with TSX

Spectre

- performs only legal memory accesses
  - has nothing to do with exception handling or suppression

# Meltdown vs. Spectre

Meltdown

- performs illegal memory accesses → we need to take care of processor exceptions
  - exception handling
  - exception suppression with TSX
  - exception suppression with branch misprediction

Spectre

- performs only legal memory accesses
  - has nothing to do with exception handling or suppression

# Meltdown vs. Spectre

Meltdown

- performs illegal memory accesses → we need to take care of processor exceptions
  - exception handling
  - exception suppression with TSX
  - exception suppression with branch misprediction

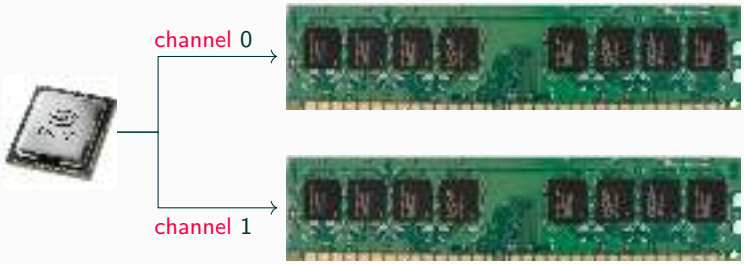
Spectre

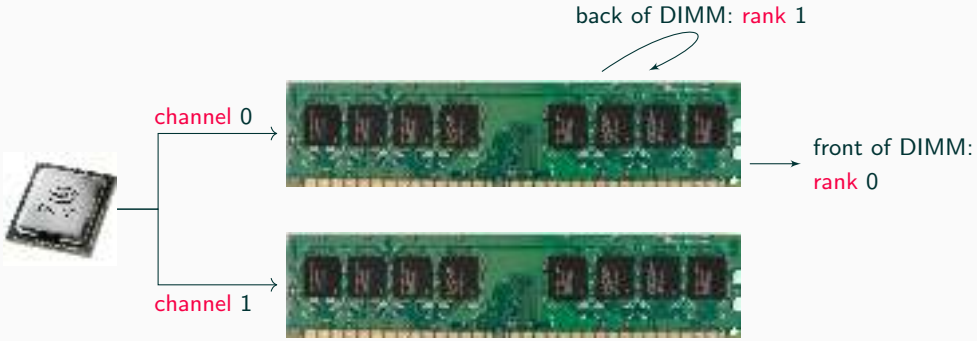
- performs only legal memory accesses
  - has nothing to do with exception handling or suppression

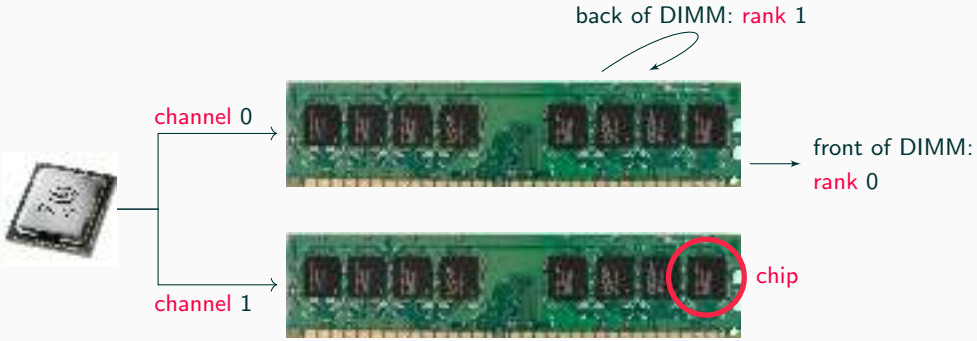
→ two papers, two names, etc.

## What if we want to modify data?

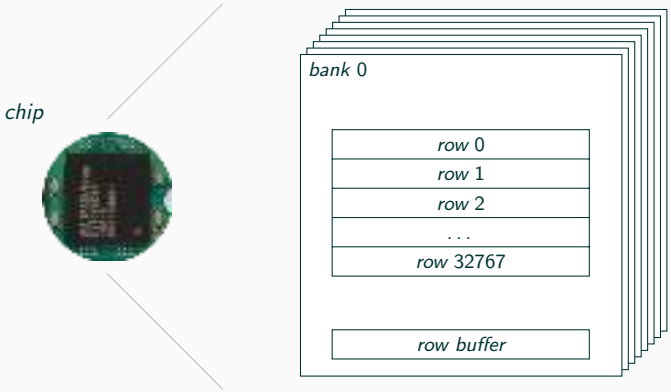
# DRAM organization



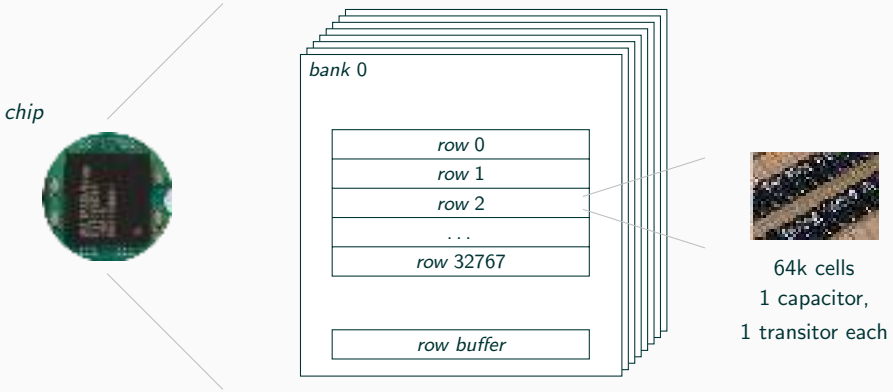

back of DIMM: rank 1
channel 0
front of DIMM:
rank 0
channel 1

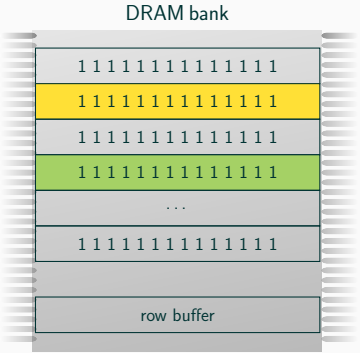

back of DIMM: rank 1
channel 0
front of DIMM:
rank 0
chip
channel 1

# DRAM organization

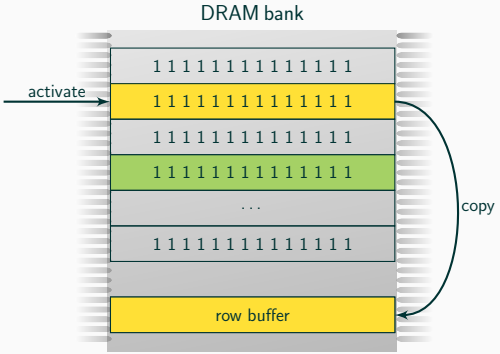


# DRAM organization

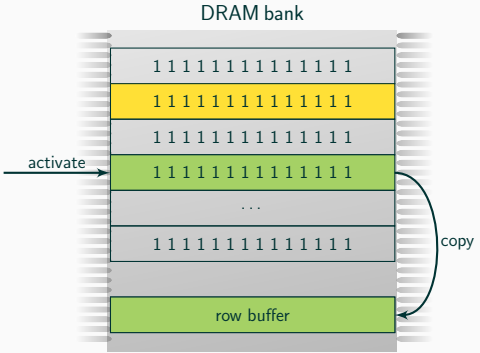




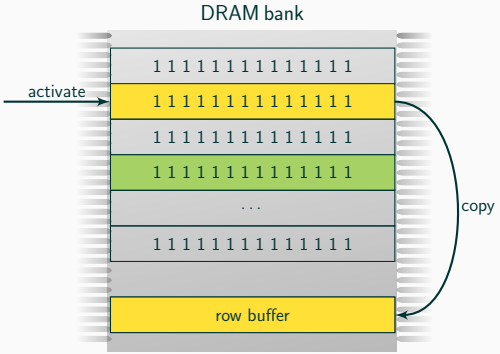

- Cells leak → repetitive refresh necessary
- Maximum interval between refreshes to guarantee data integrity
- Cells leak faster upon proximate accesses → Rowhammer



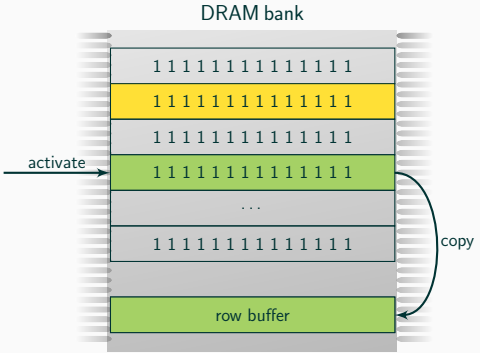

- Cells leak → repetitive refresh necessary
- Maximum interval between refreshes to guarantee data integrity
- Cells leak faster upon proximate accesses → Rowhammer



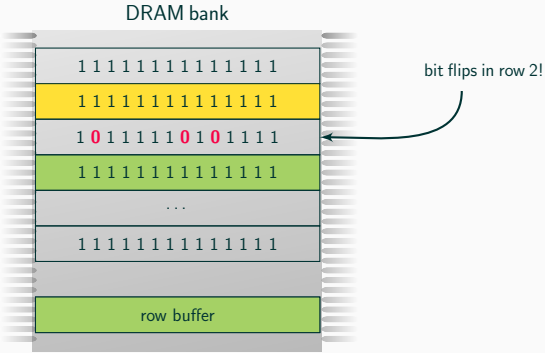

- Cells leak → repetitive refresh necessary
- Maximum interval between refreshes to guarantee data integrity
- Cells leak faster upon proximate accesses → Rowhammer




- Cells leak → repetitive refresh necessary
- Maximum interval between refreshes to guarantee data integrity
- Cells leak faster upon proximate accesses → Rowhammer




- Cells leak → repetitive refresh necessary
- Maximum interval between refreshes to guarantee data integrity
- Cells leak faster upon proximate accesses → Rowhammer




- Cells leak → repetitive refresh necessary
- Maximum interval between refreshes to guarantee data integrity
- Cells leak faster upon proximate accesses → Rowhammer

# Hammering techniques

- There are two different hammering techniques

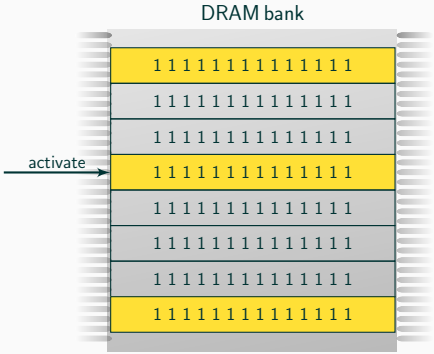
- There are two different hammering techniques
- #1: Hammer one row next to victim row and other random rows

# Hammering techniques

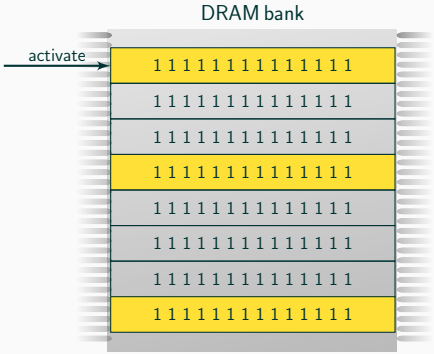
- There are two different hammering techniques
- #1: Hammer one row next to victim row and other random rows
- #2: Hammer two rows neighboring victim row

- There are **three** different hammering techniques
- #1: Hammer one row next to victim row and other random rows
- #2: Hammer two rows neighboring victim row
- #3: Hammer only one row next to victim row

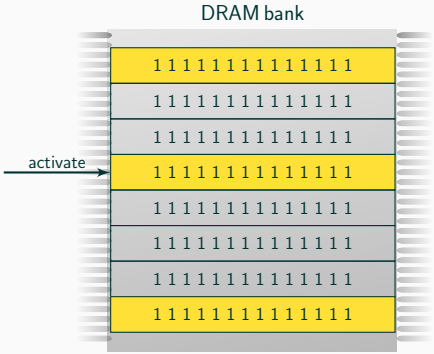
# #1 - Single-sided hammering



# #1 - Single-sided hammering

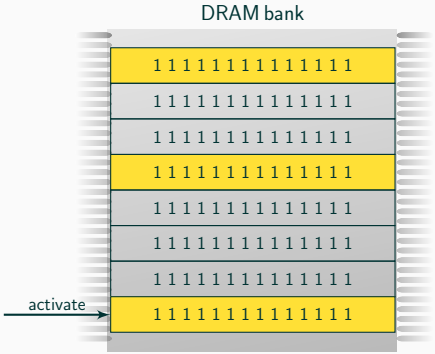


# #1 - Single-sided hammering

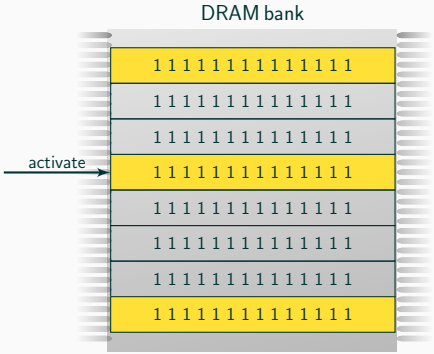


# #1 - Single-sided hammering

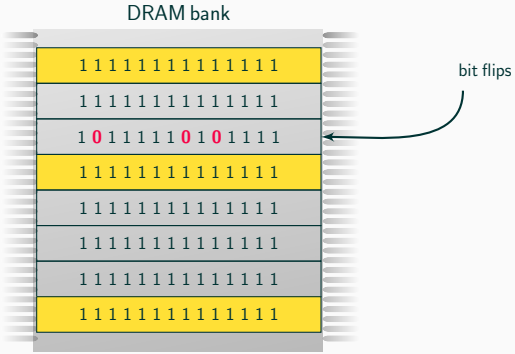




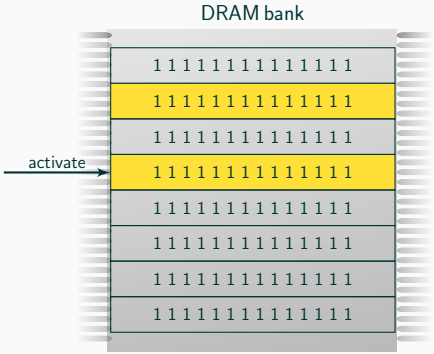
# #1 - Single-sided hammering



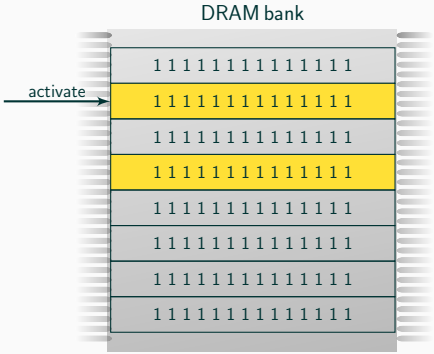
# #1 - Single-sided hammering



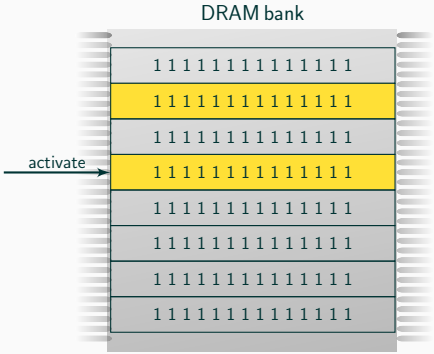
# #2 - Double-sided hammering



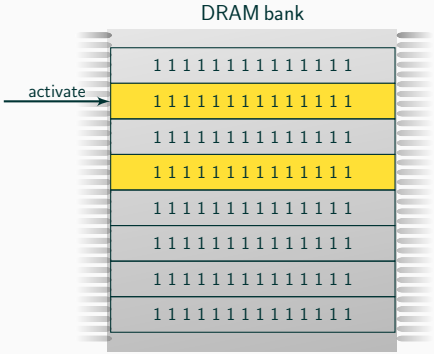
# #2 - Double-sided hammering



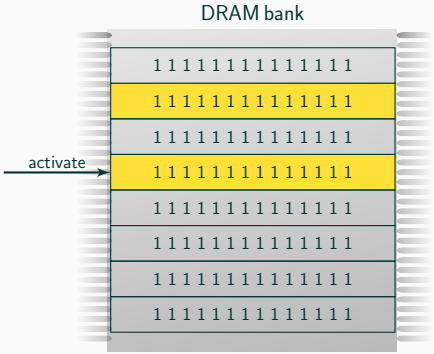
# #2 - Double-sided hammering



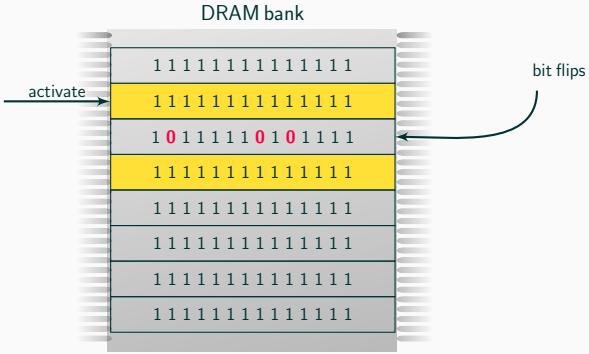
# #2 - Double-sided hammering



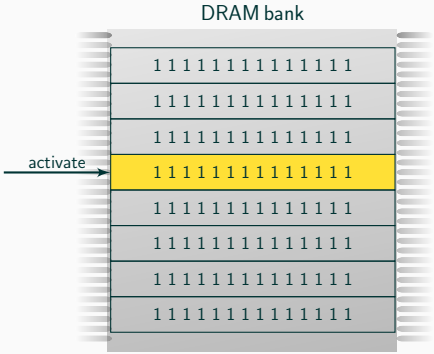
# #2 - Double-sided hammering



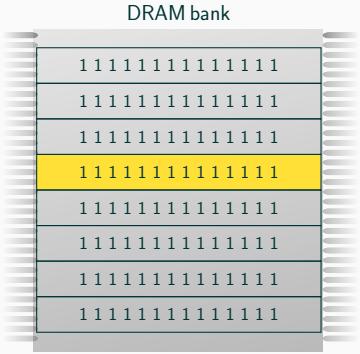
# #2 - Double-sided hammering



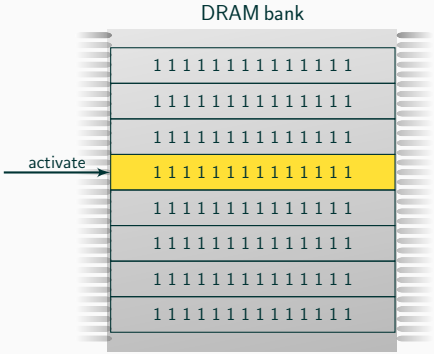
# #3 - One-location hammering



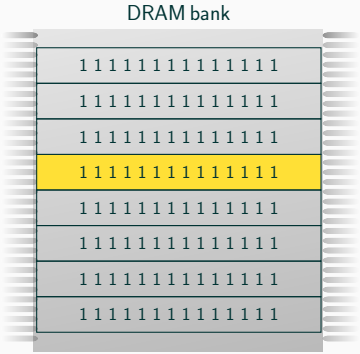
# #3 - One-location hammering



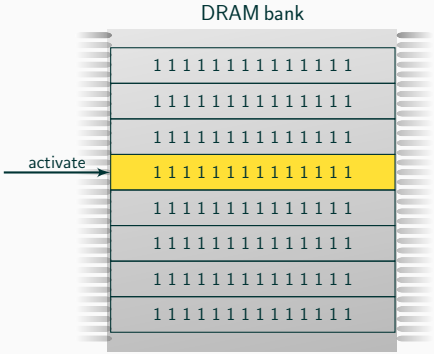
# #3 - One-location hammering



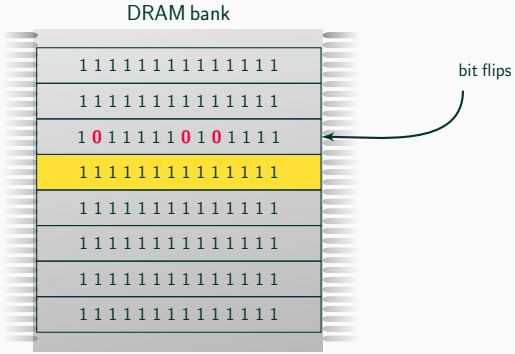
# #3 - One-location hammering



# #3 - One-location hammering



# #3 - One-location hammering



# How to exploit random bit flips?

- They are not random → highly reproducible flip pattern!
  1. Choose a (kernel) data structure that you can place at arbitrary memory locations
  2. Scan for “good” flips
  3. Place (kernel) data structure there
  4. Trigger bit flip again

# What if we cannot target kernel pages?



- Many applications perform actions as root

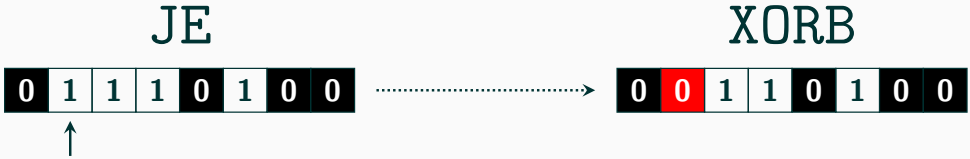
# What if we cannot target kernel pages?

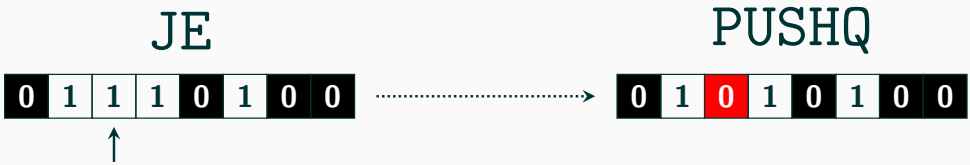
- Many applications perform actions as root
- They can be used by unprivileged users as well

# What if we cannot target kernel pages?

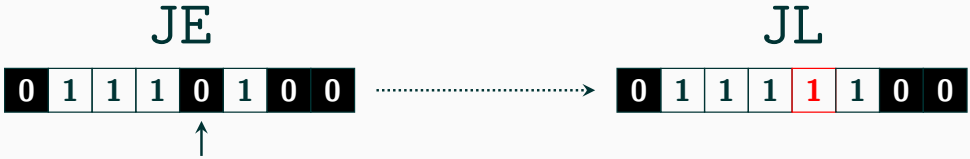
- Many applications perform actions as root
- They can be used by unprivileged users as well
- `sudo`


JE
0 1 1 1 0 1 0 0
HLT
1 1 1 1 0 1 0 0

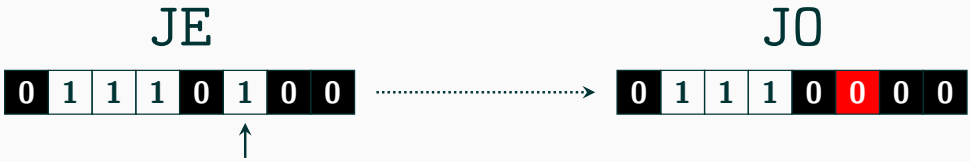

JE
0 1 1 1 0 1 0 0
XORB
0 0 1 1 0 1 0 0


JE
0 1 1 1 0 1 0 0
PUSHQ
0 1 0 1 0 1 0 0

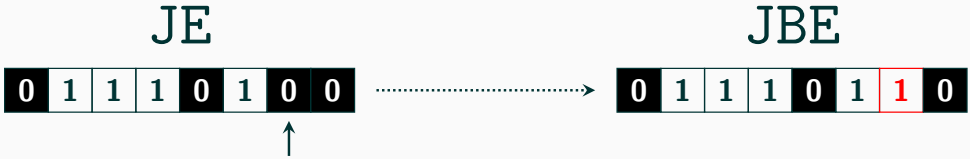

JE
0 1 1 1 0 1 0 0
<prefix>
0 1 1 0 0 1 0 0


JE
0 1 1 1 0 1 0 0
JL
0 1 1 1 1 1 0 0

# Opcode Flipping - Conditional Jump



# Opcode Flipping - Conditional Jump



# Opcode Flipping - Conditional Jump

# ECC memory vs. refresh rate

Apple had a great idea:

- lowering the refresh rate saves energy but produces more bit flips

# ECC memory vs. refresh rate

Apple had a great idea:

- lowering the refresh rate saves energy but produces more bit flips

$\rightarrow$ use ECC memory to mitigate bit flips

# ECC memory vs. refresh rate

Apple had a great idea:

- lowering the refresh rate saves energy but produces more bit flips

$\rightarrow$ use ECC memory to mitigate bit flips

- in the end: it's an optimization problem.

# ECC memory vs. refresh rate

Apple had a great idea:

- lowering the refresh rate saves energy but produces more bit flips

$\rightarrow$ use ECC memory to mitigate bit flips

- in the end: it's an optimization problem.
  - too aggressive? bit flips will be possible

# ECC memory vs. refresh rate

Apple had a great idea:

- lowering the refresh rate saves energy but produces more bit flips

$\rightarrow$ use ECC memory to mitigate bit flips

- in the end: it's an optimization problem.
  - too aggressive? bit flips will be possible
  - too cautious? waste of energy

# ECC memory vs. refresh rate

Apple had a great idea:

- lowering the refresh rate saves energy but produces more bit flips

$\rightarrow$ use ECC memory to mitigate bit flips

- in the end: it's an optimization problem.
  - too aggressive? bit flips will be possible
  - too cautious? waste of energy
  - what if the "too aggressive" changes over time?

# ECC memory vs. refresh rate

Apple had a great idea:

- lowering the refresh rate saves energy but produces more bit flips

$\rightarrow$ use ECC memory to mitigate bit flips

- in the end: it's an optimization problem.
  - too aggressive? bit flips will be possible
  - too cautious? waste of energy
  - what if the "too aggressive" changes over time?
  - what if attackers come up with slightly better attacks?

# ECC memory vs. refresh rate

Apple had a great idea:

- lowering the refresh rate saves energy but produces more bit flips

→ use ECC memory to mitigate bit flips

- in the end: it's an optimization problem.
  - too aggressive? bit flips will be possible
  - too cautious? waste of energy
  - what if the "too aggressive" changes over time?
  - what if attackers come up with slightly better attacks?

  → difficult to optimize with an intelligent adversary

# What do we learn from it?

We have ignored microarchitectural attacks for many many years:

# What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto

# What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto $\rightarrow$ "software should be fixed"

# What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto $\rightarrow$ "software should be fixed"
- attacks on ASLR

# What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto → "software should be fixed"
- attacks on ASLR → "ASLR is broken anyway"

# What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto $\rightarrow$ "software should be fixed"
- attacks on ASLR $\rightarrow$ "ASLR is broken anyway"
- attacks on SGX and TrustZone

# What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto → "software should be fixed"
- attacks on ASLR → "ASLR is broken anyway"
- attacks on SGX and TrustZone → "not part of the threat model"

# What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto → "software should be fixed"
- attacks on ASLR → "ASLR is broken anyway"
- attacks on SGX and TrustZone → "not part of the threat model"
- Rowhammer attacks

# What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto → "software should be fixed"
- attacks on ASLR → "ASLR is broken anyway"
- attacks on SGX and TrustZone → "not part of the threat model"
- Rowhammer attacks → "only affects cheap sub-standard modules"

# What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto → "software should be fixed"
- attacks on ASLR → "ASLR is broken anyway"
- attacks on SGX and TrustZone → "not part of the threat model"
- Rowhammer attacks → "only affects cheap sub-standard modules"

→ for years we solely optimized for performance

# When you read the manuals...

After learning about a side channel you realize:

# When you read the manuals...

After learning about a side channel you realize:

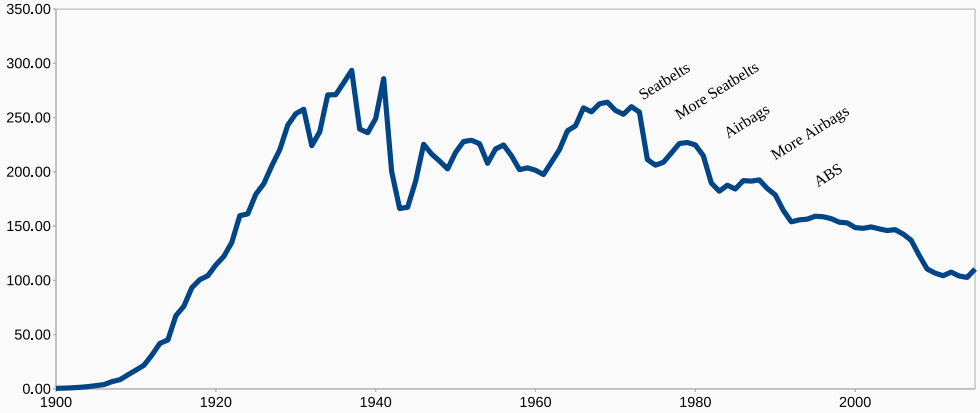
- the side channels were documented in the Intel manual

# When you read the manuals...

After learning about a side channel you realize:

- the side channels were documented in the Intel manual
- only now we understand the implications

# What do we learn from it?




Motor Vehicle Deaths in U.S. by Year

# Conclusions

A unique chance to

- rethink processor design

# Conclusions

A unique chance to

- rethink processor design
- grow up, like other fields (car industry, construction industry)

A unique chance to

- rethink processor design
- grow up, like other fields (car industry, construction industry)
- dedicate more time into identifying problems and not solely in mitigating known problems

# Microarchitectural Attacks: Meltdown and Spectre

**Daniel Gruss**

April 21, 2018

Graz University of Technology